Exploring Computational Media as a Possible Future of Software
by Marcel Borowski

A thesis presented to the Faculty of Natural Sciences of Aarhus University in partial fulfillment of the requirements for the Ph.D. degree.

S U B M I S S I O N:
February, 2022

M A I N S U P E R V I S O R:
Clemens Nylandsted Klokmose

C O - S U P E R V I S O R:
Susanne Bødker
Computational media is a vision of flexible and malleable software. Early systems like Kay and Goldberg’s Dynabook from 1977 or diSessa and Abelson’s Boxer from 1986 explored this vision of software in the past. This vision of software puts users in control of their software, makes software more understandable and programming easier, and blurs the line between development and use of software. Today’s software landscape, however, falls short of this vision and instead follows an application-centric model in which applications act as silos for functionality and data. While this enables developers and companies to compartmentalize software into smaller pieces, it also makes them the keeper of control over software. This often leaves users disempowered and out of control: Users are unable to modify their applications and functionality and are at the mercy of companies and developers.

In this thesis, I revisit computational media as an alternative to the application-centric model. Through the internet, the web, and platforms like Webstrates it is now possible to explore software that is inherently open and malleable—to create alternatives to the taken-for-grantedness of software as applications. My research focuses on the scope of the technical capabilities of computational media and the use of computational media by individual users and small groups of users. I operationalize my research through prototypes, which act as computational alternatives in empirical studies, and through the design and implementation of new platforms supporting the principles of computational media.

My research contributes to a theory, design, and empirical strand of research: Drawing from existing visions of computational media, I contribute a theoretical framing of computational media including the four principles malleability, shareability, distributability, and computability, and facets for each of these principles. Building on top of the Webstrates platform, I contribute Codestrates v2 as a development platform for Webstrates and Varv as a programming model for inherently malleable and extensible software. Lastly, I contribute findings of three empirical studies that I conducted using prototypes built with computational media, demonstrating the potentials and challenges of computational media for its users. Together, these contributions are intended to demonstrate that there are alternatives to the predominant application-centric model of software and that the strict separation of development and use of software is not set in stone but can be broken up.
RESUMÉ


I denne afhandling genbesøger jeg computationelle medier som et alternativ til den applikationsfokuserede model. Via internettet, netværk og platformer som Webstrates er det nu muligt at udforske software, der er iboende åbent og letbearbejdeligt — at skabe alternative til selvfølgeligheden i software som applikationer. Min forskning fokuserer dels på rammerne for og de tekniske muligheder inden for computationelle medier og dels på individers og mindre gruppers brug af disse medier. Jeg operationaliserer min forskning gennem prototyper, der fungerer som computationelle alternativer i empiriske studier, samt gennem design og implementering af nye platformer, der understøtter principperne bag computationelle medier.

I am very grateful for the many great people who supported me throughout my three-year long Ph.D. journey: my supervisors, collaborators, colleagues, friends, and family. It is their guidance and support that made this journey enjoyable.

First and foremost, I would like to thank my supervisors Susanne Bødker and Clemens N. Klokmose. Susanne, thank you for making my Ph.D. studies possible and for your guidance and advice during the past three years. Thank you for encouraging me in my research and a special thanks also for the support while I (almost) got stranded in Portland. Clemens, thank you for the guidance and support throughout my studies. Thanks for introducing and connecting me to other collaborators, for whiteboard sessions about prototype ideas, and more. And thanks you both, Susanne and Clemens, for creating a great working environment.

I would also like to thank my co-authors and collaborators of the papers included in this thesis. Thank you, Arvind Satyanarayan, Bjarke V. Fog, Carla F. Griggio, Clemens N. Klokmose, Ida Larsen-Ledet, James R. Eagan, Janus Bager Kristensen, Luke Murray, Midas Nouwens, and Rolf Bagge. It was a pleasure working with all of you during my studies and it helped me getting new perspectives on my work. Also thanks, Bjarke, for helping me translating the abstract of my thesis to Danish. A special thanks to Arvind for hosting me in your group at MIT CSAIL for my remote stay abroad.

I also thank all of my colleagues and fellow Ph.D. students in the CoCHI group and the UBI group for making my studies a great time. A special thanks to Banu Saatçı for encouraging me to take the step to become a Ph.D. student and for being a great friend. Also thanks, Marius Hogräfer, for sharing the same journey with me and the fun times in the Friday Bar and the Danish course. Thank you, Roman Rädle, for bringing me on the path of becoming a Ph.D. student and encouraging me to pursue this journey.

I would also like to thank my parents and my sister for their continuous and unconditional support and love throughout the years. Thank you also for all the care packages, they were always a nice surprise! Finally, Xiyu, thank you for always supporting, encouraging, and believing in me. I am grateful I could share many of the joys and sorrows of doing a Ph.D. with you. I could not have done this without the four of you!
STRUCTURE OF THE THESIS

This thesis consists of two parts: Part I provides an overview of my Ph.D. project and its contributions. It summarizes the results of my papers and describes how the projects of the papers are related to each other. Part II includes the six main papers that I contributed to during my Ph.D. studies. They consist of four peer-reviewed publications, one technical report, and one unpublished manuscript.

PART I: OVERVIEW

Part I includes eight chapters and provides an overview of my Ph.D. project. Focusing on the exploration of computational media, it provides a definition of the vision of computational media, demonstrates how software that follows this vision can be created, and presents how I used these prototypes in three cases throughout my Ph.D. studies.

In Part I, I use the pronoun “we” when reporting details or results of projects I conducted together with other co-authors. In these cases, the term refers to me and my co-authors of the respective Papers A–F. Throughout Part I of the thesis, I use small notes in the page margin, which summarize topics of paragraphs to guide the reader. An example of such a margin note is the word “Conventions” in the margin on the right side of this paragraph. Part I is structured as follows:

INTRODUCTION. Chapter 1 introduces the background and motivation of my Ph.D. project. It explains my research aim of exploring computational media, presents guiding research questions, and summarizes my research approach.

BACKGROUND AND RELATED WORK. Chapter 2 outlines the related work of my research. It, first, describes the current predominant application-centric model of software and its possible alternative computational media. Second, it summarizes related work in the fields of tailoring and malleability of software.

METHODOLOGY. Chapter 3 provides a more detailed look on my research approach and process. It illustrates the timeline of my Ph.D. project and its method triangulation using a theory, design, and empirical work strand of research. Further, it describes overarching methods and techniques that I used in order to operationalize my research: participatory design, co-design, computational alternatives, and creating possible futures.

DEFINING COMPUTATIONAL MEDIA. Chapter 4 presents my vision of computational media. This contributes to the theory strand of
my research. It, first, reviews existing visions of computational media and how they influenced my vision. Then, it describes my vision, its related concepts, and four key principles of computational media: malleability, shareability, distributability, and computability. The chapter concludes with examples of software that follows some of the principles of computational media.

**Creating Computational Media.** Chapter 5 presents the technological contributions of my research. This contributes to the design strand of my research. It, first, describes the existing Webstrates and Codestrates v1 platforms on which my works builds upon. Second, it introduces two platforms that I designed during my Ph.D. project: Codestrates v2 and Varv. It details on backgrounds of their development, presents an overview of their features, and provides insights into their implementation. Finally, the chapter analyzes each platform through the four principles of computational media.

**Using Computational Media.** Chapter 6 presents three empirical cases I worked on during my Ph.D. project. In these cases, I used computational media prototypes to explore the vision and its principles. This contributes to the empirical work strand of my research. The chapter provides a brief overview of each of these cases and summarizes the contributions in connection to computational media and its four principles.

**Reflections and Discussion.** Chapter 7 discusses the computational media vision in other contexts than the three key research questions: First, it discusses how my design work was influenced by my empirical work and lists four open challenges of these empirical cases that remain unresolved in my prototypes and platforms. Next, it relates the vision of computational media to the Common Interactive Objects framework. The chapter concludes with limitations of my research and my approach.

**Conclusion.** Chapter 8 starts by revisiting the research aims and summarizes the contributions presented in this thesis. Finally, it outlines directions for future research in the domain of computational media.

**Part II: Publications and Manuscripts**

Part II includes six papers: four peer-reviewed publications, one technical report, and one unpublished manuscript. The content of the papers is included in the state in which it was published or submitted. The only changes that were conducted as part of the process of including the papers into this thesis were fixing typographical errors and matching the typographic style to the template used in this thesis.
The remainder of this section provides an overview of the papers included. It summarizes each paper, describes my role in the projects, and how the findings are used in this thesis.


This paper is included as Chapter 9. It studies how researchers in a nanoscience lab use computational tools to conduct their research. Using a participatory design process, we developed a computational labbook prototype and used the prototype to investigate how such a medium could transform the researchers’ practice. The paper reports on three computational characteristics, which are prevalent in their practice. Finally, the paper discusses the principles of computational media.

My main role in this work was the participatory design process, the development of the prototype, and the discussion about the principles of computational media. I was, furthermore, involved in the in-situ interviews with the researchers of the nanoscience lab, and the data analysis.

This paper is the first empirical case I worked on during my Ph.D. project. In Part I, it mainly contributes to Chapter 6.


This paper is included as Chapter 10. It presents a co-design study with three stages about the topic of collaborative writing. This paper presents the overall study with three stages: (1) defining problem spaces, (2) integrating and elaborating ideas, and (3) using and reflecting on a prototype. Based on the findings of the study, the paper presents a list of contrasting needs of participants when writing collaboratively. While this paper mainly focuses on collaborative writing, its insights on contrasting needs motivate the exploration of more malleable software.

In the first two stages, my co-author Ida took the lead in designing the workshops, while I supported in conducting them. In the third stage, I prepared and developed the prototype for the participants based on their ideas from the previous stages and lead the design of the reprogramming activities.
While the paper’s findings are closer related to collaborative writing than computational media, it describes the overall structure of the second empirical case I worked on during my Ph.D. project. In Part I, it contributes to the description of the case and some findings in Chapter 6. The findings that are closer related to computational media are described in Paper C.


This paper is included as Chapter 11. It is related to the study of Paper B and focuses on the insights of the third stage, where we used a reprogrammable prototype that participants could modify during the workshops. The paper reports on observations and pitfalls in using the prototype and distills four lessons learned on how to use reprogrammable high-fidelity prototypes in prototyping workshops.

My main role in this work was the preparation of the prototype and the workshop activities, supporting participants during the workshop, and analyzing the data.

The paper describes the findings of the second empirical study that I worked on. In Part I, it contributes to Chapter 6.


This paper is included as Chapter 12. It presents the design and implementation of Codestrates v2, an iteration of the first Codestrates platform. Codestrates v2 is a development platform consisting of the three components: the Webstrates Package Manager, the Fragments and Editor Engine, and the authoring environment Cauldron. The paper introduces the components of Codestrates v2 and their implementation and demonstrates their use in two examples.

My main role in this work was to design and test the Codestrates v2 platform and to write of the technical report to document the platform.

The paper describes the first technical platform that I worked on as part of my Ph.D. project. In Part I, it contributes to Chapter 5.
This paper is included as Chapter 13. This project started by conducting a game challenge, where participants were tasked to implement a malleable and collaborative game using only the Codestrates v2 platform. Throughout the project, the paper was expanded to include a timeline of the development of Webstrates and two iterations of Codestrates. It presents tensions in the development and uses these tensions to analyze three case studies, in which we used Webstrates or Codestrates, and to analyze the data collected in the game challenge. A collection of lessons learned concludes the paper.

My main role in this work was summarizing the history of Codestrates v1 and Codestrates v2 in the timeline, identifying and formulating the tensions discussed in the paper, as well as analyzing the data of the interviews, and writing up the results and lessons learned.

The paper is the third empirical case I worked on during my Ph.D. project. In Part I, it mainly contributes to Chapter 6.

This paper is included as Chapter 14. It presents the programming model Varv, which represents interactive behavior of software as a declarative data structure. This work was motivated by the shortcomings of the JavaScript programming model, which was used previously in Webstrates and both iterations of Codestrates. The paper reports on the design of the programming model, its language structure, and example cases on how it can be used.

My main role in this work was to lead the research efforts, designing the programming model in multiple iterations, as well as supporting the development of the implementation and preparing the examples for the paper.

The paper describes the second platform that I worked on as part of my Ph.D. project. In Part I, it contributes to Chapter 5.
Besides the aforementioned papers, this subsection provides an overview of other publications that influenced my Ph.D. project and this thesis. While these are not included in this thesis, they motivated some of my work and had an influence on the redesign of Codestrates v2.


This paper was published during my Master’s. It focuses on composability and interoperability of software using a package manager to add and remove features from an application at runtime. It build on the first iteration of Codestrates and later motivated the Webstrates Package Manager of Codestrates v2.


This paper was published in the first year of my Ph.D. and presented results of a study conducted in my Master’s. The study uses Codestrates v1 in a university course for the facilitation of programming assignments. Results of this study also motivated the design of Codestrates v2.


This paper summarizes the findings of the first two cases I conducted during the first half of my Ph.D. and outlines principles and some facets of computational media.

**Part III: Appendix**

Part III includes the appendix of the papers Papers B, D, and F. The other papers do not have an appendix. The appendices are included in the same state in which they were published in the respective papers and are added for completeness.
# CONTENTS

## I OVERVIEW

1 INTRODUCTION
1.1 Research Aims and Approach ............................................. 6
1.2 Being Situated in the CIO Project ..................................... 6
1.3 Structure of Contents .................................................. 8

2 BACKGROUND AND RELATED WORK
2.1 Application-Centric Software .......................................... 9
2.2 Computational Media .................................................. 11
2.3 Tailoring ............................................................... 17
2.4 Malleability ............................................................. 20
2.5 Summary ............................................................... 21

3 METHODOLOGY
3.1 Project Timeline ......................................................... 23
3.2 Triangulation ........................................................... 26
3.3 Methods ................................................................. 28
3.4 Summary ............................................................... 30

4 DEFINING COMPUTATIONAL MEDIA
4.1 Related Visions and Motivation ......................................... 31
4.2 My Vision of Computational Media .................................... 32
4.3 Concepts ................................................................. 33
4.4 Principles ............................................................... 33
4.5 Examples ............................................................... 39
4.6 Summary ............................................................... 41

5 CREATING COMPUTATIONAL MEDIA
5.1 Webstrates .............................................................. 43
5.2 Codestrates v1 .......................................................... 46
5.3 Codestrates v2 .......................................................... 49
5.4 Varv ..................................................................... 53
5.5 Summary ............................................................... 60

6 USING COMPUTATIONAL MEDIA
6.1 Case 1: Nanoscience Lab Study ......................................... 63
6.2 Case 2: Collaborative Writing Study ................................... 68
6.3 Case 3: Game Challenge ................................................. 74
6.4 Summary ............................................................... 80

7 REFLECTIONS AND DISCUSSION
7.1 Iterative Cycles on Design and Empirical Work ..................... 81
7.2 Relationship to Common Interactive Objects ......................... 84
7.3 Limitations ............................................................ 86

8 CONCLUSION
8.1 Thesis Contributions .................................................... 89
8.2 Directions for Future Work ............................................. 91
## CONTENTS

8.3 Concluding Remarks .................................. 93

II PUBLICATIONS AND MANUSCRIPTS

9 PAPER A: NANOSCIENCE LAB STUDY .......................... 97
  9.1 Introduction ........................................ 97
  9.2 Related Work ..................................... 98
  9.3 Method .......................................... 101
  9.4 Findings ........................................ 105
  9.5 Discussion ...................................... 115
  9.6 Conclusion ..................................... 119

10 PAPER B: COLLABORATIVE WRITING STUDY .................. 121
  10.1 Introduction .................................... 121
  10.2 Related Work ................................... 123
  10.3 Co-design Study ................................ 124
  10.4 Findings ....................................... 132
  10.5 It Looks Like They Don’t Agree ..................... 138
  10.6 Discussion ..................................... 142
  10.7 Conclusion ..................................... 145

11 PAPER C: PROTOTYPING WORKSHOPS .......................... 147
  11.1 Introduction .................................... 147
  11.2 Related Work ................................... 149
  11.3 Case ........................................... 150
  11.4 Observations ................................... 156
  11.5 Lessons Learned ................................. 159
  11.6 Limitations and Future Work ....................... 163
  11.7 Conclusion ..................................... 163

12 PAPER D: CODESTRATES V2 ................................ 165
  12.1 Introduction .................................... 165
  12.2 Background ..................................... 167
  12.3 Webstrates Package Manager ....................... 170
  12.4 Execution Engine ................................ 171
  12.5 Cauldron ....................................... 174
  12.6 Examples ....................................... 177
  12.7 Implementation ................................ 178
  12.8 Conclusion ..................................... 181

13 PAPER E: GAME CHALLENGE .................................. 183
  13.1 Introduction .................................... 183
  13.2 Related Work ................................... 186
  13.3 Background: Webstrates and Codestrates ............ 190
  13.4 Realizing the Vision of Computational Media ...... 196
  13.5 Tensions in the Realization of Computational Media . 203
  13.6 Case Studies .................................... 213
  13.7 The Reprogrammable Game Challenge ............... 218
  13.8 The Games ..................................... 222
  13.9 Results ......................................... 227
  13.10 Lessons Learned ............................... 244

xviii
## CONTENTS

13.11 Conclusion ................................................. 247

14 PAPER F: VARV .......................... 249
14.1 Introduction ................................. 249
14.2 The Varv Language ............................... 252
14.3 The Varv Architecture ......................... 260
14.4 Implementation ............................... 262
14.5 Case Studies ................................. 266
14.6 Tooling ............................................. 271
14.7 Related Work ................................. 274
14.8 Discussion ......................................... 278
14.9 Conclusion ......................................... 282

### III APPENDIX

A APPENDIX PAPER B: COLLABORATIVE WRITING STUDY 287
A.1 Scenarios and Disruptions ............... 287

B APPENDIX PAPER D: CODESTRATES V2 291
B.1 Todo List Example Code .................. 291

C APPENDIX PAPER F: VARV 295
C.1 Varv Language Example .................. 295
C.2 List of Primitive Actions ................. 298
C.3 List of Primitive Triggers ................. 301
C.4 List of Data Stores ......................... 301
C.5 List of Extensions ......................... 302
C.6 List of DOM View Template Tags ....... 302

BIBLIOGRAPHY ........................................ 305
| Figure 1.1 | Applications and computational media | 4 |
| Figure 3.1 | Chronological timeline of projects | 24 |
| Figure 3.2 | Triangulation | 27 |
| Figure 5.1 | The user interface of Codestrates v1 | 47 |
| Figure 5.2 | The user interface of Cauldron | 50 |
| Figure 5.3 | Varv concept language example | 57 |
| Figure 5.4 | Architecture overview of Varv | 58 |
| Figure 6.1 | Screenshot of the computational labbook | 65 |
| Figure 6.2 | Screenshots of the prototype’s two views | 70 |
| Figure 7.1 | Overview of the connections between my empirical work and design work | 82 |
| Figure 9.1 | Applications and computational media | 100 |
| Figure 9.2 | Overview of researcher and participant engagement in the research process | 102 |
| Figure 9.3 | A lab bench in one of the main laboratories | 106 |
| Figure 9.4 | Screenshot of the computational labbook | 112 |
| Figure 10.1 | Participants’ handwritten notes and sketches | 122 |
| Figure 10.2 | Timeline of the study | 126 |
| Figure 10.3 | Overview of the stages and their outcomes | 126 |
| Figure 10.4 | The conceptual blend diagram | 128 |
| Figure 10.5 | Photos from workshops of each stage | 129 |
| Figure 11.1 | Screenshots of the prototype’s two views | 153 |
| Figure 11.2 | Screenshots of examples of modifications | 158 |
| Figure 12.1 | The software stack of Codestrates v2 | 167 |
| Figure 12.2 | A screenshot of Codestrates v1 | 169 |
| Figure 12.3 | Data synchronization of HTML fragments | 174 |
| Figure 12.4 | An overview of Cauldron | 176 |
| Figure 12.5 | The todo list example application | 178 |
| Figure 12.6 | The Ganymede computational notebook | 179 |
| Figure 13.1 | Applications and computational media | 185 |
| Figure 13.2 | Transclusion | 192 |
| Figure 13.3 | A grocery list developed using Codestrates v1 | 193 |
| Figure 13.4 | Data synchronization of HTML fragments | 194 |
| Figure 13.5 | A screenshot of a web page with Codestrates v1 | 195 |
| Figure 13.6 | The computational labbook prototype | 214 |
| Figure 13.7 | The PARTICIPATE prototype | 215 |
| Figure 13.8 | Codestrates v2 based exercise sheet | 217 |
| Figure 13.9 | Screenshots of the games | 224 |
| Figure 14.1 | Varv Examples | 250 |
| Figure 14.2 | The components of a Varv concept definition | 255 |
| Figure 14.3 | The structure of an event in Varv | 258 |
Figure 14.4 Example of an event flow in Varv . . . . . . . . . . 259
Figure 14.5 Architecture overview of Varv . . . . . . . . . . 260
Figure 14.6 Screenshots of our implementations of Varv . . 263
Figure 14.7 The software stack . . . . . . . . . . . . . . . . . . 264
Figure 14.8 Screenshots of the first case study . . . . . . . . . 268
Figure 14.9 Screenshots of other example applications . . . . 271
Figure 14.10 Summary of three authoring tools . . . . . . . . 272
Figure A.1 Scenario A and associated disruption . . . . . . 287
Figure A.2 Scenario B and associated disruption . . . . . . 288
Figure A.3 Scenario C and associated disruption . . . . . . 289

LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.1</td>
<td>Principles and facets of computational media</td>
<td>35</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Overview of facets supported by platforms</td>
<td>61</td>
</tr>
<tr>
<td>Table 6.1</td>
<td>Potentials and challenges found in the first case</td>
<td>66</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>Potentials and challenges found in the second case</td>
<td>72</td>
</tr>
<tr>
<td>Table 6.3</td>
<td>Potentials and challenges found in the third case</td>
<td>78</td>
</tr>
<tr>
<td>Table 9.1</td>
<td>Overview of principles in the computational labbook</td>
<td>115</td>
</tr>
<tr>
<td>Table 10.1</td>
<td>Overview of the participants’ attendance</td>
<td>131</td>
</tr>
<tr>
<td>Table 11.1</td>
<td>Extensions available for participants</td>
<td>152</td>
</tr>
<tr>
<td>Table 11.2</td>
<td>Overview of the the groups</td>
<td>155</td>
</tr>
<tr>
<td>Table 12.1</td>
<td>The libraries used for fragments</td>
<td>180</td>
</tr>
<tr>
<td>Table 13.1</td>
<td>Overview of the participants</td>
<td>218</td>
</tr>
<tr>
<td>Table 13.2</td>
<td>Overview of the groups</td>
<td>219</td>
</tr>
</tbody>
</table>

LIST OF LISTINGS

<p>| Listing 12.1 | The DOM structure of a codestrate                               | 170  |
| Listing 12.2 | A simple example WPM package                                    | 171  |
| Listing 12.3 | An example of how packages can be required                      | 172  |
| Listing 12.4 | Example of a JavaScript and HTML fragment                       | 173  |
| Listing 12.5 | Creating an editor for a fragment                               | 175  |
| Listing C.1 | Example of an abstract concept definition                       | 296  |
| Listing C.2 | Example of an concrete concept definition                       | 297  |
| Listing C.3 | Example of a template of a todo list app                        | 297  |
| Listing C.4 | Example action                                                   | 298  |</p>
<table>
<thead>
<tr>
<th>List of Acronyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
</tr>
<tr>
<td>CIO</td>
</tr>
<tr>
<td>CRDT</td>
</tr>
<tr>
<td>CSCW</td>
</tr>
<tr>
<td>CSS</td>
</tr>
<tr>
<td>DOM</td>
</tr>
<tr>
<td>EUD</td>
</tr>
<tr>
<td>EUP</td>
</tr>
<tr>
<td>EUSE</td>
</tr>
<tr>
<td>IDE</td>
</tr>
<tr>
<td>HTML</td>
</tr>
<tr>
<td>JSON</td>
</tr>
<tr>
<td>PDF</td>
</tr>
<tr>
<td>SCSS</td>
</tr>
<tr>
<td>SVG</td>
</tr>
<tr>
<td>URL</td>
</tr>
<tr>
<td>WPM</td>
</tr>
<tr>
<td>WYSIWIS</td>
</tr>
<tr>
<td>WYSIWYG</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>List of Videos</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Paper A</strong></td>
</tr>
<tr>
<td><strong>Paper C</strong></td>
</tr>
<tr>
<td><strong>Paper F</strong></td>
</tr>
</tbody>
</table>

(Videos retrieved February 22, 2022)
Part I

OVERVIEW
Some mass items, such as cars and television sets, attempt to anticipate and provide for a variety of applications in a fairly inflexible way; those who wish to do something different will have to put in considerable effort. Other items, such as paper and clay, offer many dimensions of possibility and high resolution; these can be used in an unanticipated way by many, though tools need to be made or obtained to stir some of the medium’s possibilities while constraining others.

— Kay and Goldberg [152]

Software as computational media, a flexible and malleable medium, is a vision that dates back to the early years of software in the 1970s. The most prominent vision likely was the one of personal dynamic media by Kay and Goldberg [152] in 1977 in which they compare a computational medium with paper or clay, which can be formed by people and appropriated in numerous ways. In his vision of the Dynabook, Kay [149] envisioned that even children should be able to modify and reprogram the games they are playing. diSessa and Abelson [78] explored a similar vision as reconstructible computational media and challenged the current view on programming at the time by attempting to make programming easy. diSessa [77] depicts applications as “monolithic” and “unmodifiable,” such as the circles on the left in Figure 1.1. Tools created with computational media, on the other hand, can be “organically enriched, altered, and combined” [77] as depicted on the right in Figure 1.1. However, both the Dynabook from Kay and Goldberg and the Boxer system from diSessa and Abelson were ahead of its time—for instance, the development machines for Boxer costing “the better part of $100,000” [77].

The current situation of software, however, looks different from those visions: Instead of computational media, software nowadays exists mostly in the form of applications. Applications act as silos of functionality that are used to fulfill a limited set of tasks. For instance, the word processor Microsoft Word can be used to write texts such as this thesis, whereas the vector graphics editor Adobe Illustrator might be less suited to do so. The latter, however, might be far superior to create figures like the ones used in this thesis.

The creation and the use of applications are most of the time strictly separate tasks: Software developers in companies design and implement the functionality, which is required to fulfill a certain set of tasks, bundle it in an application, and ship it to the end-user either
via an executable file or as a web or mobile app. The only role of end-users in this model is to use applications as turn-key, one-size-fits-all software. There is a clear separation between who is the developer of software and who is the end-user. Such an application-centric model of software brings benefits for developers and companies: Software can be developed and tested in smaller chunks, functionality can be ensured to work in this certain sandbox, and shipping software without the source code as binary files prevents pirating software.

This application-centric model, however, also poses many challenges for end-users: First, functionality is often final and prevents users from adding new functionality. While plug-ins such as browser extensions sometimes allow users to change certain parts of their applications [185], they are often difficult to create and require programming knowledge. Functionality, furthermore, is siloed in applications, making it impossible to transfer certain functionality from one application to another — an idea that the interaction paradigm instrumental interaction [25] envisioned: interaction instruments that can be used beyond their original use case on different types of objects. Also underlying operating systems can restrict the use of functionality, for instance, when applications are only available on a specific operating system. Not only functionality, also data is often bound to applications and stored in proprietary data formats that first need to be exported to open data types such as JSON (JavaScript Object Notation) to be used in other applications. These restrictions are even exaggerated in collaborative settings where collaborators have to compromise which applications to use so that all collaborators can use it [234].

The current trend of software moving towards web apps, enforced online connections to use applications, and subscription services reduce the already little control end-users have over their software even more — leaving users unable to negotiate their software [233]. Externally enforced software updates, such as in web apps, can undermine skill-based values of applications [234]. Software, further, can lock users into certain ecosystems, for instance, by preventing to export data, which can discourage users from switching to other
introduction

To regain control over software, one opportunity is to look beyond application-based software and imagine software that is more malleable from the ground up.

Consider the following example: In Google Tasks, a todo list web app that is integrated into Gmail, it is possible to create tasks in a list. Each task can have title, description, and due date. However, it is not possible to assign other users to tasks, or add labels to tasks like it is possible to emails in Gmail. To add such features would require access to the infrastructure and underlying data model of the Google Tasks web app, and like require to install additional software like code editors or even IDEs (Integrated Development Environments). In the current software model even such simple modifications are either impossible or would require significant effort.

In the vision of software created with computational media, a user should be able to make such a modification with little effort: For instance, a user could open an authoring environment directly on the web page with the todo list web app and inspect the code for the todo list. Through structured concepts and declarative actions in the code, a user could find the implementation of the todo list and modify it right away. Changes would be applied live, so that they can be directly tested. Once the modification is done, the editor can be closed and the modification is persisted within the web app and can be used on all clients of the user.

In this thesis, I explore the notion of software as computational media. The idea of computational media blurs the strict line between development and use and between software developer and end-user. The internet and the web offer a ubiquitous platform for software that works across operating systems and devices. The Webstrates [162] and Codestrates [257] platforms leverage this potential and make it malleable and collaborative. This creates a stable platform to explore the vision of computational media with practical and operational prototypes and warrants revisiting software as computational media today. I explore computational media on a theoretical, a design, and an empirical level. To operationalize my research and scope it towards a Ph.D. project, I focus my research on the technological capabilities of computational media and its use from individual users or small groups of users rather than its cultural and economical impact. My research aims to provide a technological foundation and an understanding of how computational media platforms can be created and of its capabilities. This is intended to act as a stepping stone towards future research on its possible cultural and economical impact.

While it is still unclear whether computational media is an answer to the challenges of the application-centric model, it is important to explore it: My research on computational media shows that software does not always have to come in the form of applications, it demonstrates that there are alternatives to how software can be made. The
strict line between software developer and end-user is often taken for granted as something set in stone. In this thesis, I demonstrate one alternative. Still, there are also other ways to better negotiate software [232] or make software more malleable and interoperable [287]. Hence, I do not consider computational media as a replacement for application-based software but rather as a complementary model that can live besides applications and is better suited for some use cases. In other use cases, for instance, when security and safety are highly critical, professional developers and application-based software might still be required.

1.1 RESEARCH AIMS AND APPROACH

The main research aims and direction of my project are exploration: I investigate what computational media is and can be, its potentials and challenges, and how to design systems that support this vision. I do this in the three domains of theory, design, and empirical work. The goal can be summarized in the following three research questions:

RQ 1 What is computational media and what are its principles?

RQ 2 How to design software supporting the vision of computational media?

RQ 3 What are potentials and challenges when using computational media?

Webstrates [162] and Codestrates [257] embody some of the principles of computational media by building on the web. Having worked with both before, these platforms were both fitting and pragmatic choice to build my own prototypes upon. Therefore, my prototypes are related to Webstrates and Codestrates and use them as my underlying platform. In order to explore computational media in a variety of contexts during the three-year long time frame of my Ph.D., I collaborated together with other colleagues on three case studies. Thus, the cases I worked on are diverse and do not always build on top of each other. The pragmatics of finding common themes that are interesting for both me and my colleagues played also a role in selecting these cases. I will elaborate on my approach and my methodology in more detail in Chapter 3.

1.2 BEING SITUATED IN THE CIO PROJECT

The research of my Ph.D. project was partially funded and is situated in the CIO (Common Interactive Objects) project [66]. The CIO project is based on the activity-theoretical principle that human activity is mediated by artifacts. When people collaborate, this collaboration
is also mediated through objects, for instance technology that they use, e.g., a shared document or spreadsheet. These common objects are used by people in communities, which “carry out their joint activities through artifact ecologies” [27]. Artifact ecologies are the collection of artifacts that are used by people or communities to fulfill their activities. For instance, using different devices like a desktop, tablet, and smartphone or applications. Artifacts in these ecologies are used in a multiplicity of contexts in different “configurations of people, applications, and devices” [27].

The CIO project works towards supporting “users to better understand and develop the technologies they use together” [27]. It aims to achieve this by creating a “coherent conceptual framework for interactivity that can afford a new focus on the relationship between technology and its use, as interactive, common objects that are mediators of collaborative human activity” [46]. The main objectives of the project are:

1. Develop the conception of common interactive objects in order to offer a new understanding of human-computer interaction, focusing on human control.
2. Develop support for building user interfaces in a coherent and unified framework.
3. Make common interactive objects that will empower users to better understand and develop the technologies they use.
4. Carry out ground-breaking research regarding the technological basis of common interactive objects with focus on malleability, control and shareability over time.

— Main objectives of the CIO project [66]

The above objectives of CIO emphasize the empowerment of users to both better understand and develop the software they use. One challenge of bringing users in control and allowing them to develop their software is that software is often not malleable and even for users with technical knowledge impossible to modify their tools. Here, computational media might support better ways of common interactive objects: Computational media is motivated to give control back to users on how they want to use their software (1. objective), to empower users to better understand and develop software they use (3. objective), and provides a technological basis with a focus on malleability and control (4. objective). Particularly the malleability of computational media might allow users to modify their software and the shareability to collaborate on software. The last two objectives are especially supported by my design contributions Codestrates v2 (Paper D) and Varv (Paper F). I will discuss the connection between computational media and CIO in more detail later in Part I in Section 7.2.
1.3 Structure of Contents

The remainder of Part I is structured as follows: First, in Chapter 2, I will provide the background of my project and give an overview of related work. I will describe the notion of application-centric software and summarize different views on the domain of computational media and other related domains. Further, I will provide an overview of tailoring of software and malleability.

Next, I will summarize the process of my Ph.D. project and illustrate the chronology of my cases and papers in Chapter 3. In addition to that, I will introduce methods which I used during my Ph.D. project such as participatory design and computational alternatives.

In Chapter 4, Chapter 5, and Chapter 6 — which form the core of my thesis — I will cover the three research questions. The results in these chapters are not presented in chronological order and are interdependent. For instance, by using computational media I learned about how to create it and what are principles and facets of it. Similarly, the created prototypes did feed back into the empirical work. I will present these results in the order of my research questions: First, in Chapter 4, I will define computational media including its principles and facets (Paper A), second, in Chapter 6, I will describe how to design software that supports the vision of computational media (Papers D and F), and, third, in Chapter 5, I will describe what potentials and challenges of computational media includes in use (Papers A, B, C, and E).

After the main contributions of my thesis, I will reflect on my methodology, discuss computational media in relation to the CIO project, and describe limitations of my research in Chapter 7. Finally, in Chapter 8, I will summarize my contributions and touch on directions for future research on computational media.
My Ph.D. project was influenced and motivated by multiple domains of research. I will start by presenting challenges of the current, predominant model of software creation: the application-centric model. As a possible alternative, I will then describe related work in the field of computational, dynamic, and reconstructible media, which offer an alternative way of creating software. A key principle of computational media is malleability and empowering users to change their software. Hence, I will review related work on this topic and the related field of tailorable and malleability.

As most of my empirical and design projects were conducted with other co-authors — each with their own focus of research — the related work in the included papers in Part II also focuses on other domains such as collaborative writing or declarative programming. I do not discuss these domains, which are not closely related to my Ph.D. project, in Part I.

2.1 APPLICATION-CENTRIC SOFTWARE

Rather than the computer [...] as a “clay of computing” by Kay and Goldberg (1977), the paradigmatic application model of software seems to be teaching people that a computer contains turn-key products of pre-packaged functionality that you adapt to, rather than adapting it to you.

— Nouwens and Klokmose [235]

Software is often synonymous with applications: Interactive software with a limited set of functionality to perform tasks in a certain domain. In this model, applications are often combined with files, which contain information or data. The application, then, provides the functionality to interpret, view, and modify this information or data. For instance, in a word processor application like Microsoft Word only files with certain file types like .docx can be opened and modified.

This application-model of software brings several challenges with itself. For one, applications are often one-size-fits-all and not malleable, i.e. they cannot be modified by the user, and are often difficult to combine. Furthermore, they often require negotiation of preferences when multiple people work together. Other challenges include:

RIGID FUNCTIONALITY. The functionality of applications is often rigid and cannot be modified or extended by its users. While some
software prevents any type of modification, e.g., by being closed-source, other might allow for customization through preferences or extension through plugins or scripts. A study on Danish knowledge workers found that these workers rarely personalize their software using plugins or scripts [235]. Nouwens and Klokmose [235], however, acknowledge that it is unclear whether applications are just good enough or customizing software is too difficult. Lowering the cost of customization could lead to more users making use of it.

**FUNCTIONALITY AND INFORMATION SILOS.** Functionality of applications and information in files is often siloed. This means, on the one hand, that functionality cannot be moved from one application to another, forcing users to combine multiple applications. For example, while the color picker functionality of Adobe Illustrator is superior compared to the color picker in Microsoft Word, it is not possible to move such functionality over to the other application. On the other hand, this means that information in files is often stored in proprietary formats, that cannot be interpreted by other applications than the one they were created in. Fox et al. [108], therefore, argue for need of information spaces that can access information across applications. Another related field of research to this issue is activity-based computing [19], where the activity of the user and not a particular application is the basic computational unit.

**COST OF COMBINING APPLICATIONS.** If the functionality of an application is insufficient for a task at hand, users are forced to either search for a more powerful application or to combine multiple applications into “meaningful bundles” [19]. This, however, requires reconfigurations of applications when multi-tasking. A lack of standards in data types or protocols exacerbates this problem and often requires the user to export and import compatible file types to share them across application boundaries. For instance, some of the vector figures in this thesis were created in the application Affinity Designer, which uses the file type .afdesign. LaTeX, in which this thesis was created in, however, cannot interpret this file type. To make use of it in LaTeX, the file first has to be exported in Affinity Designer to the PDF (Portable Document Format) format. Subsequent changes to the Affinity Designer file always require another export step to use the updated file in LaTeX. This translation between applications comes “at the cost of flexibility and interactivity” [234].

**PREFERENCES AND COMPROMISES IN COLLABORATION.** When collaborating with other users, the previously mentioned problems are even more salient, because the application-centric model does not only require users to combine multiple applications to solve their tasks but, furthermore, to find compromises with other users [234].
These compromises come at a cost and forces collaborators to use their “lowest common software denominator: the application that was still available after the various constraints were taken into account (device and OS compatibility, privacy restrictions, price, interpersonal skill levels, etc)” [234]. This forces users to abandon their preferred applications and practices.

I use the challenges of the application-centric model as a motivation for my research on computational media. They motivate some of the principles and facets of computational media and design decisions in my prototypes.

The origin of the application-centric model dates back to the initial days of computing and modes of creating and distributing software. For instance, shipping applications in binary code limits the access of customers, prohibiting unlicensed modifications of software [232]. While related to the application-centric model, my Ph.D. project focuses on the vision of computational media, how it can be realized, and its implications—not the origins of the application-centric model.

2.2 COMPUTATIONAL MEDIA

Computational media is a vision of malleable and computational software and ways to realize it. Over the years, different terms such as personal dynamic media [152], reconstructible computational media [78], or shareable dynamic media [162] have been used to describe a similar vision of a malleable and dynamic software medium. In this thesis I use the term computational media to refer to this vision of software. Computational media is an alternative to the application-centric model in that it combines functionality and information in the same document.

In this section, I will summarize related work in this field including software visions, research prototypes and commercial products. I use the term “computational media” as a set of principles, which can be used as a lens to investigate the qualities of a system or framework and not as a categorical term of a certain type of software systems. I elaborate on my use of the term in Chapter 4.

2.2.1 Early Visions

Early visions of computational media date back to the 1970s, when Kay [149] introduced his vision of the Dynabook in 1972. Kay envisioned the Dynabook to be a thin and tablet-like device with a built-in keyboard that can be used as a personal computer. Least as impressive as his vision for the hardware of the Dynabook at that time was his vision for its simple and malleable software: With the Dynabook, even children should be able to reprogram and modify the functionality of videogames that they can play on the device.
Five years later, in 1977, Kay and Goldberg [152] presented their first functional prototype of this vision: the *Interim Dynabook*. While its hardware was still a shelf-sized desktop computer and not yet a slim and mobile tablet, its software was build on Smalltalk (see below) and created a malleable and computational playground in which small applications could be created and modified after they were created. These applications ranged from text editing, to drawing, to animation and simulation. Kay and Goldberg coined their vision of such a medium *Personal Dynamic Media* and compared it to the properties of physical materials like paper or clay that “offer many dimensions of possibility” and “can be used in an unanticipated way by many” [152].

Another early vision of computational media was implemented in the Boxer system by diSessa and Abelson [78]. Boxer was built on the two main principles *spatial metaphor* and *naive realism* to make programming easier. It “challenges, in a small way, the current view of programming languages” [78]. In Boxer, applications consisted of text and boxes—the latter were able to be nested inside each other. Details in boxes could be hidden while still being available for inspection when needed. Using boxes, applications could be structured in a way where the “box containment reflects meanings such as subprocedures as parts of procedures” [78].

The vision of diSessa and Abelson, which they named *Reconstructible Computational Media*, aimed at lowering the barriers of programming for nonexperts to control their software. It was designed for “naive and novice users” [76] and should “serve beginners and casual users” [78]. In 2001, diSessa, further, described that applications are “monolithic” and “nonmodifiable,” whereas a “computational medium allows seeding with small but extendible tools […] these tools can be organically enriched, altered, and combined, as successive layers” [77] (see Figure 1.1).

### 2.2.2 *HyperCard and Compound Documents*

A popular system that followed a similar vision of a malleable platform was Apple’s HyperCard. HyperCard provided an environment that was structured into stacks. Each stack contained multiple cards, which, again, contained objects like labels, buttons, and graphics. Cards could be linked to each other through objects and objects could be made interactive by using the HyperTalk scripting language. HyperTalk was focusing on a simple and “English-like” programming language, that made the system accessible for less technical users like school children [311]. After around ten years after its initial release in 1987, HyperCard was discontinued by Apple. According to its creator Atkinson, this was due to not being seen as “essential” anymore during a time in which Apple was struggling commercially [12, 45:47]—hence,
it was cut. Another reason might have been that the web was emerging and followed similar ideas as HyperCard with web pages as cards—supporting similar objects and links like HyperCard. The web also was cross-platform, while HyperCard was only available on the Macintosh, and made it easier to distribute web pages, whereas stacks of cards in HyperCard were stored locally on a computer. However, the web was lacking one essential part of HyperCard: an integrated authoring environment—contrary to cards in HyperCard, web pages are mainly used as a consumer and not as an author.

Microsoft OLE [221] (Object Linking and Embedding) and Apple OpenDoc [9] were attempts to break up the strict silos of functionality of applications in the 1990s. Microsoft OLE’s goal was to create compound documents that “seamlessly integrate various types of data, or components” like “[s]ound clips, spreadsheets, and bitmaps” [221]. OLE was a proprietary standard from Microsoft. OpenDoc attempted to create an open standard that could be used across platforms. OpenDoc, however, had many challenges in doing so: On the one hand, due to its component structure it required more memory than what was usual at the time in Macintosh computers and did not create an economy around components [205]. On the other hand, also the upcoming web might have shifted interest away from OpenDoc. Eventually, similar to HyperCard, also OpenDoc was abandoned by Apple in 1997—probably due to similar reasons of not being considered “essential.”

Still, some of the ideas of components, which can be used across different applications and files, are reemerging: For instance, Microsoft Fluid [222] was announced in 2021 and aims to bring Fluid components to Microsoft Office. Fluid components can, e.g., contain spreadsheets, documents, or todo lists, and can be placed in documents (Microsoft Office Suite), chats and video conferences (Microsoft Teams), emails, or a whiteboard. Content in Fluid components is synchronized with a server and enables collaborative editing in real-time. Similarly, web apps often allow for embedding or linking content into one another. However, embedded content in the web is often only possible on the level of a whole document—it is not possible to embed individual functionality or tools from other websites as OpenDoc envisioned when using components.

2.2.3 Smalltalk and Self

Smalltalk is a programming language and reprogrammable software environment that was developed in the 1970s at Xerox PARC. It was created to explore Kay’s vision of “a personal computer for children of all ages” [149]. The first practical implementation of Smalltalk was Smalltalk-72, which was—as the name implies—created in 1972 [139, 150]. This first version of Smalltalk was mainly used internally within
Xerox PARC, e.g., to explore the teaching of “object-oriented programming to middle school and high school students” [139]. Through the 1970s four more iterations of Smalltalk were created until, in 1980, Smalltalk-80 was released as the first generally available version of Smalltalk. The latter was more refined and standardized, and included a documentation—all of which leading to “significant adoption in industry and academia” according to Ingalls [139].

Smalltalk follows the main principle that “everything is an object” [139] and that these objects communicate via messages with each other—Smalltalk values simplicity. Smalltalk, further, is immediately responsive, i.e. changes to the implementation code are directly applied—even while debugging code. This is possible, because Smalltalk is implemented in itself: even the debugger and primitive methods such as the mathematical operations + and - are implemented in the Smalltalk language and can be modified and overwitten by users. Early versions of Smalltalk featured drawing, sound, and animation capabilities. Later versions added features like overlapping window interfaces, simulation, and a class browser and a debugger [139].

Besides Smalltalk, Self [298, 299] is another programming language, which focused on creating a simple yet flexible language that allows users to reprogram applications at runtime. Smith and Ungar [278] describe their vision of Self as follows: “the Self programmer lives and acts in a consistent and malleable world.” This means, that each “visual element in Self, from the largest window to the smallest triangle is a directly manipulable object” [278]. The Morphic [207] user interface environment of Self was later also added to Squeak [139].

Continuing the efforts of creating the Smalltalk language, Squeak was published in 1997 as a “open, highly-portable Smalltalk implementation whose virtual machine is written entirely in Smalltalk” [138]. One motivation for this was to bring Smalltalk to more platforms and to make it widely accessible: Running in a virtual machine allowed Squeak to be ported to Macintosh, several UNIX platforms, Windows 95, and Windows NT, crossing boundaries between operating systems. Besides portability also malleability was a focus of Squeak and Ingalls et al. [138] aimed to make “it easy to debug, analyze, and change.” This effort was continued almost another decade later by the Lively Kernel, which lifted the Squeak platform into the web and made it possible to create and modify applications in the web browser [140]. To support easy sharing of so-called parts in Lively Kernel, Lincke et al. [184] later created the Lively PartsBin, which acted as a cloud-based repository for parts.

Overall, Squeak had mostly a research focus and the “goal of producing an exploratory software system” [139]. Pharo [36] followed the footsteps of Squeak as an open source implementation of the Smalltalk programming environment. It strived to resolve licensing issues of Squeak and to provide a more lean and modern platform.
with an “emphasis on stability and commercial support” [139]. As a fork of Squeak, it is also highly portable and written entirely in Smalltalk. Glamorous Toolkit [112] is built in Pharo and follows the goal of a novel development experience that allows “people to make the interior of systems explainable” [112]. It focuses on a “moldable development” and describes its programming style as “programming in the inspector.” This makes it possible for users to change the view for each individual object and use a fitting visualization to inspect it.

While Smalltalk and Squeak have been used widely in programming education research, e.g., in projects like Squeak Etoys [151] or Scratch [259]. Ingalls [139] emphasizes that there is still a small community of users, that use Smalltalk, Squeak, or Pharo. While successful in these educational use cases, Smalltalk struggled to become commercially successful. The exact reasons for this are unclear, however, according to Wirfs-Brock [312], a central challenge of commercial Smalltalk was the shift from “fat clients” — the problem, which the commercial version of Smalltalk aimed to solve — towards thin clients due to the emergence of the web. Combined with the push of Sun Microsystems to establish Java as the solution for both web and desktop applications, this shifted the interest away from Smalltalk, made Smalltalk’s use decline, and finally caused it to never be “revived” [312]. Still, Wirfs-Brock [312] also mentions that this was not the reason Smalltalk was created in the first place: “Smalltalk wasn’t created to rule the software world, it was created to enable the invention of a new software world.”

2.2.4 Webstrates and Codestrates v1

More recently, Klokmose et al. [162] explored the vision of computational media with what they call Shareable Dynamic Media. They extended the personal dynamic media vision of Kay and Goldberg [152] by adding a collaborative and cross-device aspect to it: Instead of only being malleable, shareable dynamic media should also be shareable, i.e. being used in collaboration seamlessly, and distributable, i.e. documents could be opened and moved across heterogeneous devices. They exemplified their vision by creating the Webstrates [162] platform, which builds on web technologies and persists the DOM (Document Object Model) of a website with a server. Webstrates, thus, implements classic hypermedia virtues [54, 288], is document-centric, and makes the DOM “a first class citizen of a hypermedia system” [54].

Using Webstrates as its underlying platform, Rädle et al. [257] created Codestrates v1 as an authoring environment inspired by computational notebooks (see next subsection). This made it possible to create web apps directly in the web browser without the need for

---

1 For clarity, I refer to the initial version of Codestrates by Rädle et al. [257] as “Codestrates v1” throughout this thesis.
the developer tools. A package management [52] was later added to make sharing of functionality across documents easier and the platform composable.

Throughout the years, Webstrates and Codestrates v1 were used in a variety of projects, including collaborative data visualization [16, 135, 210], public libraries [119, 318] collaborative video editing [164], collaborative programming assignments [53], hybrid meetings [118], and affinity diagramming [191].

I will expand on the Webstrates and Codestrates v1 platforms in Chapter 5. I will describe the platforms in more detail and explain how I use them to build research prototypes — the computational labbook (Paper A) and the malleable word processor (Papers B and C) — and new platforms — Codestrates v2 (Paper D) and Varv (Paper F).

2.2.5 Computational Notebooks and No-Code Environments

Two other related types of software to computational media are computational notebooks and no-code environments. Computational notebooks are authoring environments that are motivated by the literate computing [248] paradigm. They interleave “text with code and results to construct a complete piece that relies equally on the textual explanations and the computational components” [248]. Platforms include computational notebooks like Jupyter [166], Observable [236], Google Colaboratory [116], or Codestrates v1 [257]. Computational notebooks can follow different goals like programming for computation, programming for interaction, and programming for software engineering [105]. One of the tasks computational notebooks is used for most often is data analysis and visualization [105, 154, 267]. They allow users to create computational narratives that can be published as interactive publications like, for instance, on platforms like Distill [80]. Also others have explored ways to create interactive articles on the web [67]. However, computational notebooks can also aim for different goals: Codestrates v1, for instance, is focusing on software engineering rather than computation and data analysis [105].

The other genre, no-code environments, aims to enable computational features without the hurdle of having to program. Notion [231], for instance, makes it possible to create customized product roadmaps, calendars, Kanban boards, and more without the need for programming. Simple building blocks and templates make it possible to combine functionality in idiosyncratic ways and match them to the use case at hand. Other examples with similar functionality are Coda [64], Airtable [2], and Microsoft Loop [220].
2.3 TAILORING

One key motivation of computational media is to make software malleable so that users can modify and tailor their software to their idiosyncratic needs. Hence, this subsection is dedicated to this topic and summarizes related work on tailoring and tailoring cultures—a field that goes beyond the domain of computational media. Finally, this section will briefly discuss end-user development as a related field to tailoring. The next section will then focus on malleable software.

2.3.1 Tailoring Process

Tailoring describes the process of users adapting applications to their idiosyncratic needs. For instance, changing the keyboard shortcuts of an application to use functions faster, changing the aesthetic color theme of an application, or adding extensions to a web browser. Tailoring is important as it is “impossible to design systems which are appropriate for all users and all situations” [197]. While I am using the term tailorable software in my thesis to refer to this research field, others have also called it customizable [199, 200] or personalizable [125, 126]. The field of end-user development [182] and meta-design [102], which I discuss in more detail below, share similar concerns about users’ “mismatches between their needs and the support of that an existing system can provide for them” [102].

Tailoring of software can happen on multiple levels. Mørch [225], for instance, defined the three levels: customization, i.e. changing the appearance or predefined configurations of an application, integration, i.e. recording a sequence of and storing it as a new command like a macro, and extension, i.e. changing the source code of an application. MacLean et al. [197] use the term tailoring power to order these levels: changing parameters in a configuration is less powerful than reprogramming the source code of an application, because the latter offers more possibilities in which the software can be changed.

Each of these levels of tailoring requires different levels of technical skill from users. These skill levels range from being able to simply change parameter values in a settings menu to having professional programming skills. This was coined the tailorability mountain by MacLean et al. [197] and describes that higher tailoring power usually requires higher skills from a user. A problem of this tailorability mountain is that it is often steep and that there is usually a gap between tailoring with lower skill and higher skill levels. For instance, in the web browser users can change the default font family for the whole browser by changing a parameter in the settings. But when users want to change the default font family for individual websites they need to either search for an existing extension or create their own extension, which usually requires installing an IDE, programming skills, and
deploying the extension in their own web browser. There is a large gap in between these two options. Also Mackay [201] reported that 33% of the 51 users, who participated in her study on customization in Unix software environments, mentioned that the hard difficulty of modifying software is a barrier for them to do so. In the system Buttons, MacLean et al. [197] attempt to flatten the tailorability mountain and enable users to use a variety of tailoring techniques, each increasing slightly in tailoring power and tailoring skill.

2.3.2 Tailoring Cultures and More Capable Peers

In the previous subsection, I discussed the need for more tailoring techniques to support users with different tailoring skills. While this is an important aspect of successful tailoring activities, a culture in which “users feel in control of the system and in which tailoring is the norm” [197] is just as important. MacLean et al. [197] describe that users should be able to take advantage of the different levels of skill that different people in a community provide. For instance, in the work context, a so called handyman [197] can act as a bridge between workers without programming skills and programmers, communicating the needs of workers to programmers. Also Mackay noted that her study on sharing customizations “has demonstrated that customization is not a purely individual activity” [199].

This notion of a more capable peer, e.g., a co-worker with more technical or programming skills, was also discussed by others: While MacLean et al. [197] called them handyman, Mackay [199] called them translators, Gantt and Nardi [110] gardeners, Trigg and Bødker [294] tailors, and Fogli and Piccinno [106] end-user developers. To improve software systems, these more capable peers should be cultivated and taken into account when designing software [110]. A more capable peer can also support the development of software, for instance, in participatory programming [180] or cooperative prototyping [42].

The above mentioned research mostly focused on tailoring in the workplace during the 1990s, when software was created and used differently than today — a time “when computers were primarily desktop machines used in office work, provided and maintained by the workplace” [287]. More recent research on online customization sharing ecosystems by Haraty et al. [126] shed light into more the sharing of customizations in the 2010s. In their study, they identified different roles of users including the packers, who create customization packs for specific tasks by using existing customizations — similar to a more capable peer.
2.3.3 *End-User Development*

Lieberman et al. [182] defined EUD (End-User Development) as “a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify, or extend a software artifact.” Hence, it focuses on “empowering end-users to develop and adapt systems themselves” [182]. This includes being able to modify systems, which has already been designed and created by others — EUD considers modifications during the entire software lifecycle [20, 57]. EUD activities range from customization, for example, through editing parameters, to the creation and modification of software, for example, through incremental programming, programming by example, or rule-based programming [20, 182]. While the former customization activities are closer to the domain of tailoring, the latter step into the EUP (End-User Programming) domain.

Ko et al. [167] define EUP as “programming to achieve the result of a program primarily for personal, rather public use.” In contrast to EUD, which also focuses on modifying existing software, EUP is mainly concerned with enabling “end users to create their own programs” [57]. EUP, thus, is a subset of EUD [57]. Activities of EUP include, e.g., the above mentioned rule-based programming, programming by demonstration, or also text-based programming [57]. Ko et al. [167] emphasize that the important distinction of EUP to professional programming is its underlying intent: While professional programmers implement software mainly for other users and to earn money, the intention of EUP is to personally use the created software and not to share it with others.

Lastly, EUSE (End-User Software Engineering) focuses on “activities that address software quality issues” [167]. It follows the same goal as EUP and aims to enable end-users to create their own software but does so with a focus on software qualities like “reliability, performance, maintainability, reusability, privacy, and security”[57]. Activities and techniques include supporting end-users in identifying requirements of software and specifying it or in testing and debugging their software [57, 167].

Beyond technological tools, Fischer et al. [100, 101] envisioned the future of EUD as a “cultural transformation rather than only as a technology to create software artifacts” [100]. Through frameworks such as meta-design [102] the control of designers should be shifted to end-users and empower them to “contribute their own visions and objectives.” Software should be designed for change and enable co-creation in collaborative processes [102]. This, in the long term, could shift the “consumer cultures” towards “cultures of participation” [101] in which also end-users can contribute to the design of the software they use. Requirements for enabling such a culture include, among
other things, making changes seem possible, making them technically feasible, and lowering the barriers to share changes [101]. My research on computational media shares some of these goals and tries to create a technological platform for inherently extensible software.

2.4 Malleability

As discussed in the previous section, being malleable, dynamic, or reconstructible is a central principle of computational media. In this section, I will discuss the notion of malleable software and the interaction model instrumental interaction. While tailoring describes the process of modifying one’s software or applications, malleability describes a property of software that can be modified and “molded” by its users.

2.4.1 Malleable Software

Malleable software is software that can be modified by users to their own needs. Malleable software can support tailoring activities by providing tailoring techniques to a user. Cabitza and Simone [59] deconstruct the term “malleability” and describe its features as versatility, interoperability, and extensibility. Also Tchernavskij focuses on interoperability and (re-)combination and describes that “malleable software should dissolve the boundaries of discrete systems such as apps, letting users and developers pull them apart and (re-)combine them” [287]. Richter and Riemer, on the other hand, emphasize flexibility and openness of software “supporting a wide variety of work practices without the need for technical customization” [260]. The notion of opennable applications goes beyond software that is open only on a single level to being able to “unpack their applications (almost) at any level of detail” [59]. Related terms are also adaptable software by Trigg et al. [295], which they describe as flexible, parameterized, integratable, and tailorable. While similar to tailorable software, malleable software goes beyond being just tailorable but also interoperable or flexible. Malleability is connected to EUD in that malleable software allows for EUD activities to be facilitated, e.g., by making changes to software possible and providing mechanisms and openness to change interactive behavior.

Whether software is malleable for a user, however, depends not solely on the technical property of the software but also on the technical skill of a user—similar to the tailoring power of MacLean et al. [197]. For instance, making software reprogrammable makes it malleable for programmers but not for people with little or no programming skills.
2.5 Summary

2.4.2 Instrumental Interaction

Instrumental interaction [25] is an interaction model which describes user interfaces as domain objects and interaction instruments. Domain objects are the objects of interest of a user, i.e. objects that are being manipulated. Interaction instruments, on the other hand, are the mediator between a user and a domain object. Instruments can be used by the user to modify objects and can be compared to tools like a hammer in the physical world. Instruments can also be reified [28], i.e. can also act as domain objects themselves. In such a case other instruments can be used to modify the reified instrument, for instance, if a hammer is broken it can be repaired using other instruments like glue and acts as the object in this interaction.

The VIGO (Views, Instruments, Governors, and Objects) architecture [163] appropriated instrumental interaction to multi-surface environments. Instrumental interaction also inspired the design of Webstrates [162] in which transclusion should make it possible to transclude other webstrates as instruments into a document. In the ONE (One is Not Enough) project [26], a similar conceptual model consisting of substrates, instruments, and environments was proposed. It also emphasized interoperability between instruments and substrates, where “instruments can work with substrates even if they were not designed for each other” [26].

2.5 Summary

In this chapter, I presented related research to my work on computational media. First, I provided an overview of the notion of the application-centric model of software. I summarized challenges of application-based software that motivated my exploration of computational media. Second, I summarized related work on computational media, ranging from early visions of Kay and diSessa to the shareable dynamic media vision by Klokmose et al., which my work builds on. Third, I presented related work on tailoring and customization of software — a key principle of computational media. Fourth, as a related topic to tailoring, I discussed malleable software, which follows similar motivations as computational media.

The work I am presenting in this thesis is situated within these domains. In Chapter 4, I adopt the three principles of shareable dynamic media, extend it with a fourth principle, and present facets on how these principles can be realized in computational media platforms. This contributes a theoretical foundation of the vision of computational media. In Chapter 5, I build upon the Webstrates and Codestrates v1 platforms to create two new platforms that follow the principles of computational media. These platforms contribute towards the domain of malleable software, EUD, and how a programming model for inher-
ently extensible software could look like. Lastly, in Chapter 6, I utilize existing tailoring knowledge to create prototypes in two case studies that contribute towards tailoring practices of users and challenges of current platforms of computational media in use.
In this chapter, I will outline the methodology of my Ph.D. project. First, I will present a project timeline of the projects I conducted during my Ph.D. studies. This will provide a chronological overview of the different projects I was involved in. Second, I will discuss the triangulation of my research across three strands of research: theory, design, and empirical work. This will present an overview of the research focus of each of my projects. Lastly, I will present overarching methods I used in these projects and argue for the suitability of these methods for the explorative nature of my Ph.D. project.

3.1 PROJECT TIMELINE

This section provides a chronological overview of the projects I worked on during my Ph.D. project. Some of these projects are interrelated and their date of publication does not always reflect the order in which they were conducted. Thus, this section provides a short summary of how these projects were initiated and how they unfolded. Figure 3.1 provides an overview of my projects and resulting papers.

The goal of this section is to provide an overview of the timeline of projects I have worked on. Hence, projects are only described in brief so that they can be situated in the overall picture. Each project, including their methodology and results, will be discussed in more detail in the next chapters.

3.1.1 Initial Case Studies

As my research was motivated by exploration and my Ph.D. project was limited to three years of research — which also included teaching and course work — I decided to start my research by collaborating with colleagues on two case studies. This allowed me to explore computational media in two very different use cases.

The first of these two case studies was the Nanoscience Lab Study, which resulted in Paper A. This project was initiated by my co-authors Midas Nouwens and Clemens Klokmose through observations and interviews of a local nanoscience lab of Aarhus University. They investigated challenges in relation to using computation for their scientific work. When I joined the project, we decided to explore how computational media could be used to overcome some of these challenges. We did this through the creation of a computational labbook prototype based on Webstrates and Codestrates v1.
Figure 3.1: A chronological timeline of the projects and papers of the thesis.
The second case study was the Collaborative Writing Study, which resulted in Papers B and C. This case study was initiated by Ida Larsen-Ledet and I in order to explore current issues of collaborative writing and how more flexible tools through computational media could help to overcome these. The study was a three-stage co-design study in which we, first, defined problems spaces, second, created ideas to mitigate these problems, and, third, let users use and reflect on their ideas. For the third stage we created a malleable word processor based on Webstrates and Codestrates v1. The prototype was used in cooperative prototyping workshops where participants themselves could modify their prototypes.

3.1.2 First Platform

Influenced by these initial case studies, I started the design work that focused on advancing the platform that we used for computational media itself: Webstrates and Codestrates v1. This design work resulted in Codestrates v2 and in Paper D. Codestrates v2 aimed to improve the usability and stability of the experimental Codestrates v1 platform and to overcome some of its issue, which we identified in the first two case studies. While I lead the design of the new iteration Codestrates v2, the implementation was mostly conducted by the programmers Janus Bager Kristensen and Rolf Bagge, which work at the CAVI [58] center of Aarhus University.

3.1.3 Third Case Study

After creating the Codestrates v2 platform, I decided to use it in another case study to gain insights on how the changes would be perceived by users and whether some of the issues of Codestrates v1 were successfully resolved. I did this in a Game Challenge that was initiated by Bjarke Fog and Clemens Klokmose. In the game challenge, participants had to create reprogrammable multiplayer games using only Codestrates v2. Through interviews we gathered insights into their use and understanding of the platform. This project resulted in Paper E.

3.1.4 Second Platform

The results of the third case study echoed some of the issues of the second case study: the insufficient computational model of JavaScript for reprogrammable and malleable computational media. In this project, I set out to overcome this issue and explored declarative ways of defining interactive behavior, which resulted in the Varo platform and Paper F. Similar to the Codestrates v2 platform, the implementation work was supported by Rolf Bagge and Janus Bager Kristensen. We
evaluated the system by demonstration through the use of examples—a user study is planned for future work.

3.2 **Triangulation**

In my Ph.D. project, I conducted theory, design, and empirical work. Figure 3.2 shows an overview of the work of my papers and existing work I build on for each of the three strands of research. The strands and the diagram are inspired by work from Mackay and Fayard [204], which use the three component disciplines “Theory,” “Design of Artifacts,” and “Observation” in their model. Like the last section, this section only provides a brief overview of my work on each of these strands and will be discussed in more detail in the next chapters.

### 3.2.1 Theory Strand

In the theoretical strand, my Ph.D. project builds on the vision of computational media. Especially, the shareable dynamic media [162] vision that defined the three principles malleability, shareability, and distributability. While this vision was conceived before I started my project, my projects feed back into it based on findings from my case studies and design work. In Paper A, I contribute the fourth principle computability to the existing three principles. As part of my doctoral colloquium paper [47] and this thesis, I also contribute a more fine-granular set of facets for each of the principles of computational media. These contributions are presented in Chapter 4.

As part of my case studies, I also contributed to other theory that is not directly related to my Ph.D. project and, hence, I do not discuss in Part I: In Paper A, my co-authors and me provide insights into the computational characteristics of the nanoscience lab that we investigated and how they facilitate the use of computation in their everyday work. In the second case study, consisting of Papers B and C, we contribute contrasting needs in collaborative writing, propose a trade-off protocol, and present lessons learned from using a reprogrammable prototype in prototyping workshops. Lastly, in Paper E, we describe tensions and lessons learned of efforts in realizing the vision of computational media in functional prototypes.

### 3.2.2 Design Strand

I contribute to the design of computational media in my Ph.D. project with two platforms and two prototypes: In my platforms and prototypes I build upon Webstrates [162] and Codestrates v1 [257], which I will discuss in relation to the principles and facets of computational media. In Paper D, I report on the updated system architecture and authoring environment Codestrates v2, which was motivated by work
Figure 3.2: Triangulation: Blocks’ colors indicate the chapter they are discussed in; gray blocks are other outcomes that I do not discuss in Part 1. The letter in the corner refers to the paper that is related to that block. Blocks with a dashed border refer to existing work [162, 257], which I build upon. The block “Principles and Facets” refers to the facets of principles of computational media I condensed during writing this thesis.
I conducted prior to my Ph.D. studies [53] and Papers A–C. By creating the declarative programming model Varv I, further, contribute to the design of computational media in Paper F. These contributions are discussed in Chapter 5. While these are contributions to the overall design of computational media, in Papers A–C, I also designed concrete prototypes using computational media: the computational labbook and the malleable word processor prototypes.

3.2.3 Empirical Work Strand

Empirically, my main contributions are three case studies of using computational media prototypes and platforms in different contexts. These are captured in the Papers A, B, C, and E. In the first case, Paper A, I investigated how computation is used in a local nanoscience lab and how computational media could act as an alternative to malleable but brittle scripts and stable but rigid applications. In Papers B and C, I used computational as a platform for a reprogrammable prototype that was used in a cooperative prototyping workshop as part of a co-design study. And in Paper E, I used Codestrates v2 in a game challenge, where participants collaboratively implemented multiplayer games in Codestrates v2. These contributions are presented in Chapter 6.

3.3 Methods

In this section, I will summarize the methods that I used in my projects and that influenced my Ph.D. overall. Following an explorative approach, I used mainly qualitative methods and emphasized the involvement of participants in my case studies. Hence, I relied a lot on participatory design and co-design in the first two case studies. Section 3.3.1 will focus on these domains. As computational media is in many ways different from common application software, I also consider some of my prototypes as computational alternatives and as ways to uncover possible futures of software. Section 3.3.2 will explain these notions in more detail. I will describe the particular methodology of each of the three case studies in more detail in Chapter 6.

3.3.1 Participatory Design and Co-Design

A key principle of participatory design is that people have a basic right to make decisions about how they do their work and, indeed, how they perform any other activities where they might be supported by an IT artifact.

—Wagner [303]
Participatory design aims to empower users to be active stakeholders in the design process of their software. Popular in the Scandinavian scene—and in my research group and among my supervisors—participatory design encourages researchers and developers to collaborate with end-users as experts in their field. Techniques can range from so-called future workshops, to mock-up design, to cooperative prototyping \[43\]. In an organizational context, participatory design also has a moral and political commitment at including users in all design decisions throughout a project \[43, 303\]. The notion of co-design or co-creation follows a similar approach: It also changes the classic role of designers, researchers, and users as separate actors with separate responsibilities in the design process to a co-designing process where these roles are merged \[270\]. Co-design, however, is following less strongly the political motivations of participatory design.

Related to participatory design, the cooperative prototyping technique emphasizes that participants should be active actors in the process of designing prototype and can help to actively improve it \[43\]. Bødker and Grønbæk \[42\] highlight the importance of using a prototype, opposed to just watching a demonstration or reading a description—they call this envisionment. For instance, malleable software like HyperCard could be used in cooperative prototyping workshops to enable modifications to the prototype while using it. This makes it possible to change prototypes not only in between prototyping sessions but also in a session together with participants of the workshop \[41, 42\].

In my projects, I employed participatory design in Paper A, and co-design and cooperative prototyping in Papers B and C. In Paper A, we conducted an iterative participatory design process together with scientists from a nanoscience lab of our university. We chose participatory design, because we wanted scientists to have a say in the design of the computational labbook prototype and we ourselves wanted to better understand the practice of the scientists and how they used scripts and applications in their workflow. Meeting with the scientists for several times also allowed them to get a better picture of what our computational media prototype would be capable of and how it is different from their regular turn-key applications and self-made scripts.

In Papers B and C, we involved participants in a three-stage co-design process. We employed participants as experts of their practice \[270\] and aimed to give them agency in the design of the final prototype. During the three stages, we first defined problem spaces of collaborative writing together with participants, second integrated and elaborated ideas for tools to mitigate these problems, and third let them use and reprogram a prototype in which we implemented their tool ideas. In the third stage, we facilitated cooperative prototyping \[43\] together with participants. Our goal was to enable participants to actively shape and reprogram a high-fidelity prototype themselves.
3.3.2 Computational Alternatives and Possible Futures

Related to participatory design, the term computational alternative is used by Korsgaard et al. [168] to emphasize the need of both methodological and technical contributions and innovations. They regard participatory design not just as a mere “toolbox” but as “a tradition in the strongest sense” and emphasize that a “computational alternative is not designed to showcase a new technical solution to a well-known problem, but to elucidate problems in the otherwise taken for granted” [168]. Computational alternatives allow to investigate “alternative futures.” This notion of taken-for-grantedness is strongly related to my research, in that computational media challenges the taken-for-grantedness of software as turn-key applications. Salovaara et al. [269], likewise, talk about “possible futures.” By concretizing a vision into a particular prototype, the vision can be made “empirically researchable in the present world” [269]. A prototype therefore is used to learn about possible futures.

In Paper A, we used the computational labbook as a computational alternative. Our aim was to create a prototype for participants to experience an alternative future of software, between scripts and applications, that they could form and change themselves, while retaining user interfaces and convenience. Our computational labbook establishes a possible future and allows us to investigate how software based on computational media would work in such a scenario and how it would affect the practice of scientists, and lets scientists reflect on their current practices that are taken for granted.

The use of interactive prototypes was important for my Ph.D. project, as computational media abandons many of the current assumptions about software as applications—for example, by being inherently malleable and shareable—and, thus, requires concrete representations for users to experience in order to better understand its potentials and challenges.

3.4 Summary

In summary, I worked on five projects during my Ph.D., which can be split into three case studies—the nanoscience lab study (Paper A), the collaborative writing study (Papers B and C), and the game challenge (Paper E)—and two platforms—Codestrates v2 (Paper D) and Varv (Paper F). In these projects, I explored computational media through the three strands theory, design, and empirical work. My method and research approach focused on qualitative research and the involvement of participants into the development of prototypes. Through prototypes that acted as computational alternatives I aimed at establishing possible futures of software.
In this chapter, I am focusing on defining computational media. I, first, provide a retrospective on the use of the term “computational media” and other related terms such as “dynamic media” or “reconstructible media.” Next, I describe my use of the term and present four key principles of computational media. This chapter, hence, is related to my first research question about what are principles of computational media and contributes to the theory on computational media. Parts of this chapter are based on my progress report and doctoral colloquium paper [47].

4.1 RELATED VISIONS AND MOTIVATION

As described in Section 2.2, the vision of a computational medium dates back to the work of Kay and Goldberg [152] on personal dynamic media and the Dynabook prototype. Kay and Goldberg envisioned a medium that is malleable like paper or clay and can be changed by any user. For example, Kay [149] described the vision of two children sitting in the grass and playing a game on their tablet-like Dynabook. They could edit the code of the game live, right in their Dynabook, and change the behavior of the game to add new features. The focus of personal dynamic media, therefore, was to make software malleable and enable any user—including even children—to modify their software and “handle virtually all of its owner’s information-related needs” [152].

Next, diSessa and Abelson [78] used the term reconstructible computational medium to describe their vision of a reprogrammable software medium. diSessa and Abelson emphasized the need for a reconstructible medium to be easy to program and allow people to reconstruct the medium to build their personalized tools. They demonstrated their vision by implementing the Boxer system. The focus of reconstructible computational media was to make programming easier and give control over a reconstructible medium also to “nonexperts” and “ordinary people” [78].

In a report from 2014, Wardrip-Fruin and Mateas [306] use the term computational media to describe new types of digital media that are enabled through computation, for instance, media art, video games, and computer graphics. They, however, do not limit the outcomes of computational media to artifacts but also include connected capabilities, insights, and educated practitioners. Wardrip-Fruin and Mateas’
use of the term “computational media,” hence, is broader than the use of Kay or diSessa. Inspired by the term “dynamic media” Klokmose et al. [162] used the term shareable dynamic media to describe a medium that is not only malleable but also shareable and distributable: Shareability enables not only to share documents with others but also real-time collaboration and distributability allows it to distribute documents across device boundaries. This vision of computational media is the closest to my vision and also the vision which I directly build upon.

The Malleable Systems Collective [206], recently, also described their vision of malleable software and systems, which also follows similar goals to making software less rigid and editable by the end-user. For instance, they advocate for software to be “easy to change” and that users should “retain ownership and control” of their software.

4.2 MY VISION OF COMPUTATIONAL MEDIA

In my work, I use the term “computational media” as a set of principles that describe what a computational medium should enable and how it could function. These four principles are then complemented by different facets, that describe more concrete manifestations of these principles and how they can be realized. Both the principles and facets can be used as a constructive tool when designing computational media systems or platforms but also as an analytical and critical lens when investigating the qualities of a computational media system or platform. Hence, computational media is not a categorical term for types of software — i.e., it does not categorize software as being a computational medium or not being a computational medium.

The four key principles of my definition of computational media are: malleability, shareability, distributability, and computability. These four principles allow for a variety of features of software when applied individually. Together, however, they complement each other and allow for new ways of creating and using software in contrast to the currently predominant application-centric model. This list of principles, however, might not be final: In the same way as I found new facets of each of the principles during my case studies, I imagine future work to possibly uncover yet new facets or even principles of computational media.

My use of the term is mostly related to the vision of shareable dynamic media by Klokmose et al. [162] but also relates to the work of Kay and diSessa. Also the notion of malleable systems by the Malleable Systems Collective [206] has overlapping motivations and principles to my work. While, for instance, diSessa and Abelson [78] envisioned even “ordinary” people without programming skills to modify their software, my Ph.D. project focuses on creating a foundation in which software can become malleable at all. This means that my design work — Codestrates v2 and Varv (Papers D and F) — focuses
on creating malleable software platforms for users with some technical and programing knowledge. This is intended as a step towards the grander long-term vision of making software malleable also for end-users without programming skills.

In the remainder of this chapter, I explain the four principles and different facets of them. Finally, I clarify how these principles can be combined to create even more powerful software.

4.3 CONCEPTS

As part of my definition of computational media I define two concepts that are used as part of the principles of computational media:

**COMPUTATIONAL MEDIUM.** A computational medium is a software platform in which documents (see below) can be created. It is the “clay” of computational media from which documents can be formed. For instance, Webstrates [162] or Jupyter [256] can be considered computational mediums.

**DOCUMENT.** A document is a single packaged entity of a computational medium. It inhabits both data and functionality within the same space — akin to a substrate [26, 162]. This follows a document-centric model [54]. For instance, a single webstrate is a document in Webstrates and a single computational notebook is a document in Jupyter. I derive the term “document” from the document-centric model and chose it over “substrate” to simplify the term.

4.4 PRINCIPLES

As computational media acts as a vision of software, I will describe principles that such a vision of software should inhabit. The principles malleability, shareability, and distributability were initially defined by Klokmose et al. [162] in the original Webstrates paper. I build on these principles and define multiple facets for each of these principles. Furthermore, I add the principle computability as part of my first case study (Paper A). The facets of computational media, which I present in this chapter, are both informed by the three case studies I worked on and which are presented in Chapter 6, and existing related work, e.g., from the domain of tailoring. Table 4.1 provides an overview of the four principles and their facets.

4.4.1 Malleability

Malleability enables users to change and shape their software to their own idiosyncratic needs. The principle of malleability goes back to the first visions of computational media like dynamic media [152] or
reconstructible media [78], and was also discussed in the vision of shareable dynamic media [162]. It can be achieved in various ways, it can range from simply adding new pre-packaged plug-ins to existing software, to changing configuration parameters, to reprogramming the source code of a system. The paragraphs below summarize five facets of how malleability can be achieved in a system. These facets are also inspired by related work on tailoring [e.g., 197, 225].

While this principle and the facets below describe different ways of moving towards malleable software, malleability is always also connected to the expertise of the user. As discussed in Section 2.3.1, software that is malleable for one type of user might not be for other types of users.

**Extensibility.** Extensibility allows users to extend and reduce the functionality of a document in the form of packages or add-ons. Adding and removing packages should be possible at runtime and changes should be directly applied. Examples for extensibility are browser extensions or code editor extensions that can be added from an extension repository or marketplace.

**Configurability.** Configurability enables to configure existing functionality through parameters or predefined options. This allows users to customize their software without the need for programming skill. However, this facet of malleability is usually limited to the available parameters and options that a software creator provides.

**Reprogrammability.** Reprogrammability provides users with tools to view and edit the underlying code of functionality within a document. Software can be reprogrammable at different levels, e.g., using a block-based programming language like Scratch [259] or Blockly [115] or by changing the source code in a code editor. Depending on this level and the planned modification users need more or less programming skills to make use of this facet of malleability.

**Copyability.** Copyability enables users to create copies of their documents. This can allow for experimentation and prototypical development of applications. For instance, one document can act as a “prototype” and users can copy this prototype including its implementation code and perform changes on their personal copy. In collaborative scenarios this could, for instance, also make it possible to create personalized views and tools while working on the same underlying data with collaborators.

**Interoperability.** Interoperability makes it possible to use the same functionality and tools with different types of data. A computational medium should provide mechanisms to support interoperability
4.4 PRINCIPLES

<table>
<thead>
<tr>
<th>PRINCIPLE</th>
<th>FACETS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malleability</td>
<td>- Extensibility</td>
</tr>
<tr>
<td></td>
<td>- Configurability</td>
</tr>
<tr>
<td></td>
<td>- Reprogrammability</td>
</tr>
<tr>
<td></td>
<td>- Copyability</td>
</tr>
<tr>
<td></td>
<td>- Interoperability</td>
</tr>
<tr>
<td>Shareability</td>
<td>- Addressability</td>
</tr>
<tr>
<td></td>
<td>- Real-time Synchronization</td>
</tr>
<tr>
<td></td>
<td>- Support for Asynchronous Work</td>
</tr>
<tr>
<td>Distributability</td>
<td>- Distributable Documents</td>
</tr>
<tr>
<td></td>
<td>- Distributable Functionality</td>
</tr>
<tr>
<td></td>
<td>- Distributable Computation</td>
</tr>
<tr>
<td></td>
<td>- Decentralization</td>
</tr>
<tr>
<td>Computability</td>
<td>- Embedded Computational Environment</td>
</tr>
<tr>
<td></td>
<td>- Code Authoring and Execution</td>
</tr>
<tr>
<td></td>
<td>- Polyglot Programming</td>
</tr>
<tr>
<td></td>
<td>- Matching Programming Model</td>
</tr>
<tr>
<td></td>
<td>- Addressable User Interface</td>
</tr>
</tbody>
</table>

Table 4.1: Principles and facets of computational media.

between functionality created in it. This is related to the interaction model instrumental interaction, which proposed a model where interaction instruments are interoperable across different types of domain objects. Ways towards interoperability, for instance, are using open data types or improving compatibility of data through “edit lenses” [186], which translate data for different tools.

4.4.2 Shareability

Shareability enables software to be shareable with other users and enables collaboration. This principle was introduced in the shareable dynamic media [162] vision. Shareability was motivated by the challenges users face when sharing documents or collaborating on documents with other users [162]. However, shareability enables users not only to collaborate together to solve a task, but also to receive or give support to other users as a more capable peer, as employed by the participants in my first case study (Paper A). The following paragraphs summarize three facets of shareability:
Addressability. Addressability makes documents addressable through a link or URL (Uniform Resource Locator). Being able to address a document in this way makes it easy to share it with other users by simply sharing the link to the document. This prevents having to send files back and forth when collaborating with others. A document in a web app like Google Docs, for example, usually is addressable via an URL.

Real-time synchronization. When a document is opened by multiple users or clients, e.g., a web browser on a device, the content of documents should be synchronized in real-time. This real-time synchronization acts as a foundation for real-time collaboration and awareness in documents. To support real-time collaboration in a meaningful way is dependent on the use case and task at hand. Especially research in the field of CSCW (Computer-supported Cooperative Work) can help to inform this facet.

Support for asynchronous work. Besides supporting real-time synchronization, a computational medium should also provide support for asynchronous work. While real-time collaboration can be a blessing when working together in some situations, in other situations it might be a curse, for instance, when interfering with other users or in cases with not enough awareness [84]. Also in my second and third case studies (Papers C and E) and previous work [53] this came up as problem when programming collaboratively. Hence, a computational medium should provide mechanisms to seamlessly switch between synchronous and asynchronous collaboration while collaboratively working on a document. Especially for collaborative coding tasks synchronous collaboration can cause novel types of conflicts such as the execution of unfinished code that is currently being edited by a collaborator (Paper E).

4.4.3 Distributability

Distributability describes that software should be distributable across a variety of heterogeneous devices and operating systems. Functionality should not be tied to a particular device type or operating system. Like shareability, distributability was introduced by Klokmose et al. [162] in their shareable dynamic media vision and was, further, motivated by ubiquitous computing [308]. The following paragraphs summarize four facets of distributability:

Distributable documents. The first and most obvious facet of distributability is that documents of a computational medium should be distributable across different device types and operating systems. For instance, regular applications or apps are often tied to a device
factor, e.g., desktop computer or a smartphone, or operating systems, e.g., Windows or iOS. Even when applications exist across different device types or operating systems, they vary in the functionality they provide and, for instance, mobile apps can have less functionality compared to their desktop counterparts. Documents of a computational medium should run across all these platforms and provide the same functionality across each of these. A prominent example of a platform that fulfills these needs is the web: almost each desktop computer, laptop, tablet, or smartphone provides a web browser to access websites — something also Webstrates [162] leverages.

**Distributable Functionality.** Not only documents as a whole should be distributable but also individual parts of their functionality and user interfaces. This enables cross-device interactions where, for instance, a laptop can be used to edit a text document and a tablet to display the preview of the document.

**Distributable Computation.** When using computation in a document, for instance, to analyze data or render 3D models, this computational load should also be distributable across different devices. This makes it possible to use the computational power of more powerful devices: For instance, in the data analysis tool Vistrates [16] the computation required for the data analysis of large amounts of data can be offloaded from a mobile laptop computer to a desktop computer with a more powerful processor to compute data faster and without draining the battery on the laptop. Another example is offloading computation to a server as done in Videostrates [164]. In my first case study (Paper A), we also made use of this ability and ran scripts on a server to speed up the nanoscientists’ scripts.

**Decentralization.** Ideally, the process of distribution should be facilitated decentralized. This makes it possible to use documents offline without internet access to a central server. Furthermore, it can prevent software feeling slow due to delays through the server connection — something which has also been discussed in the context of so-called local-first software [161].

4.4.4 Computability

Lastly, as the name implies, computability describes that a computational medium should support custom computation from within a document. Computation is an essential part of computational media, as it enables to create custom functionality within documents. Computability is related, yet different, from malleability: Malleability describes the process of editing functionality of software in a development mindset, e.g., changing the action of an existing button.
Computability, however, describes the process of adding custom computation to the center of using software, for instance, using formulas in spreadsheet applications or scripts in computational notebooks. The following paragraphs summarize facets of computability in a computational medium:

**Embedded Computational Environment.** The computational environment of a document should be part of the document itself and independent from the client. This prevents incompatibilities of code across heterogeneous devices and operating systems — similar to the benefits of using a virtual machine across different clients. Otherwise, for example, different versions of a Python distribution might lead to errors on different clients. Another example of this are the different case sensitivities of commands in operating systems: In my first case study (Paper A), participants used scripts that were written in a case-insensitive operating system in their own case-sensitive operating system. This sometimes lead to errors due to the wrong capitalization of commands in the scripts.

**Code Authoring and Execution.** Code authoring and execution should be possible from within a document. A user should not be required to install additional software like an IDE or code editor to make use of computation. Again, spreadsheets and computational notebooks offer good ways of integrating code authoring and execution into a document.

**Polyglot Programming.** Computational media should support the use of multiple programming languages, i.e. be polyglot. Beyond just supporting to run scripts of different languages in the same environment, it is also important to support the interplay between these scripts, so that, for instance, methods from one language can be exported, imported, and used in another language.

**Matching Programming Model.** Computational media requires a matching programming model that inherently supports change and malleability. As a medium that is malleable, i.e., can be modified at runtime, shareable, i.e., allows real-time collaboration of multiple users, and computable, computational media needs a programming model that fits these needs.

Many imperative programming languages, e.g., JavaScript, are not designed to support changes to running code or collaborative editing of code. This can cause a variety of issues that I explored in my case studies (Papers C and E). For instance, re-running JavaScript code does not update existing methods or event listeners in memory but instead merely adds new event listeners on top of the old ones. When collaborating, the version of code running in memory on one client can...
also differ from the version in another client, causing incompatibilities and requiring users to restart the runtime by reloading the website.

How exactly a matching programming model looks like, however, is still an open question. In the Varv project (Paper F), I explored the use of declarative data structures as a way to define interactive behavior, which resolved some of these issues but opens up new challenges to authoring code.

ADDRESSABLE USER INTERFACE. The user interface of a document should be addressable. For instance, buttons or fields should be addressable by code, so that users can use them in their computations. This is similar to the vision of Self, where each visual element should be manipulatable [278]. They should be reifiable [28], so that their interactive behavior can be changed using other tools or code. Ideally, the computational medium should also support the user in making connections between the user interface elements and the underlying code that generated them or adds their behavior. This is similar to work by Tchernavskij [287], who called out the issue of programming environments not helping to “trace graphical elements and their behavior to the code producing them.”

4.5 EXAMPLES

In this section, I will give some examples of types of computational media. First, I will discuss the Smalltalk and Squeak platforms. Second, I will cover the platforms I worked with during my Ph.D. project: Webstrates and Codestrates v1. Next, I will discuss computational notebooks, which especially in recent years gained popularity for their use in data analysis and exploration tasks [154]. Lastly, I will discuss computational media that is less focusing on programming: spreadsheets and no-code environments. These examples, further, illustrate how the principles of computational media can be used as an analytical lens for software.

4.5.1 Smalltalk and Squeak

Smalltalk and its Squeak [138] implementation focus highly on malleability. As mentioned in Section 2.2.3, Squeak is written in itself: This means that the implementation of all Squeak classes and objects is accessible from within the Squeak environment. Hence, the system is highly reprogrammable and allows users to even modify the implementation code of tools like the class browser or the debugger. To make use of this reprogrammability, however, users are required to have programming skills and knowledge about Smalltalk systems to navigate its implementation—a challenge that is exaggerated by not many people having programming knowledge in Smalltalk as
opposed to, for instance, JavaScript. Squeak also supports copyability due to the runtime state of a Squeak environment being stored as a single image file that can be copied. Providing tools like a playground, Squeak also emphasizes computability and using computation in tools. Similar to state, also the computational environment is embedded into the image and, hence, prevents incompatibilities when using the same image file across devices.

The shareability and distributability of Smalltalk and Squeak, however, is limited: While the image files can be transferred to different devices, the Squeak environment needs to be installed on all devices. The Squeak environment also is not shareable: it is a development system that runs locally on a machine and does not support real-time collaboration. While Lively Kernel overcomes the distributability of image files and allows access in the web browser, it still does not support real-time collaboration within the same environment.

4.5.2 Webstrates and Codestrates v1

Webstrates [162] primarily aims to support the first three principles: malleability, shareability, and distributability. It achieves this by making web pages persistent as webstrates. Being accessible in the web browser across devices and allowing for real-time collaboration makes them distributable and shareable. Being able to change the underlying code of applications right within the web browser using the developer tools, further, makes applications reprogrammable — albeit requiring a high technical skill level of users. Webstrates, however, did not focus as much on computability and allowing the execution of code as part of using an application.

Codestrates v1 [257] builds on these existing principles and adds an a code execution model similar to that of computational notebooks by following the literate computing paradigm [248]. This allows to execute custom code as part of using the application, making it computable.

4.5.3 Computational Notebooks

Computational notebooks such as Jupyter [256] or Observable [236] feature some principles of computational media. They, first and foremost, feature computability as a central principle by allowing to execute custom computations in order to, for example, analyze and visualize data. Some platforms, like Observable, also feature distributable by being accessible and running in a web browser. In addition, Observable also allows for shareability by allowing users to share notebooks as an addressable URL and edit them in real-time together.

While featuring computability, many computational notebooks are not malleable, as they do not allow users to change the functionality of a notebook itself from within a notebook. Additional tools like IDEs
are required to change their code base and compile new versions of the notebook themselves. Lastly, some computational notebooks are closed-source and prevent users from modifying them at all.

4.5.4 Spreadsheets

Spreadsheets are one of the most common computational mediums. They make it possible to use computation through snippets of code in cells. However, this computability is often restricted to pre-defined commands that can be used in cells — and those are often not editable. Distributability and shareability of spreadsheets strongly depends on the implementation at hand: For instance, while Google Docs support both distributability across different devices and shareability with real-time collaboration, the desktop application of Microsoft Excel (excluding Office 365) supports neither.

4.5.5 No-Code Environments

Yet another computational medium are no-code environments that allow to use computation without requiring programming skills. For instance, Notion [231] or Coda [64] allow users to create Kanban boards or custom calendars. Notion, for instance, follows a block-based model, where documents consists of blocks with different functionality, for instance, a header block, a paragraph block, or more complex blocks like a gallery or database block. These can be combined in idiosyncratic ways to create new ways of using computation. Notion also provides script-like capabilities using commands such as @Today to automatically add the current date to a paragraph [105]. Being accessible in the web browser and collaborative, platforms like Notion are also providing distributability and shareability.

4.6 SUMMARY

In this chapter, I introduced my vision of computational media. I revisited existing visions and defined how my vision of computational media is inspired by those and can act as an analytical lens. I discussed the four principles malleability, shareability, distributability, and computability, and their respective facets (see Table 4.1 for an overview). These principles and facets illustrate what a computational medium should aim and strive for and can be used as an analytical lens for software. Lastly, I illustrated examples of computational media: Squeak, Webstrates and Codestrates v1, computational notebooks, spreadsheets, and no-code environments. In these examples, I demonstrated using the principles and facets as an analytical lens. In the next chapter, I will discuss my efforts to create working experimental prototypes that aim to realize the vision of computational media.
In this chapter, I will introduce multiple experimental research prototypes that attempt to realize the vision of computational media. First, I will present the Webstrates [162] and Codestrates v1 [257] platforms, which both were developed before I started my Ph.D. project. During my Master’s studies, I used both Webstrates and Codestrates v1 for my Master’s project. I developed a package manager for Codestrates v1 [52] and used it in a study during an introductory interaction course [53]. Thus, being familiar with these platforms, I decided to focus on them as the technological foundation for my Ph.D. project. This was, further, motivated by the level of malleability Webstrates and Codestrates v1 provided in contrast to computational notebooks or spreadsheets.

During my Ph.D project, I conducted two case studies with the Webstrates and Codestrates v1 platforms (Papers A–C). Building on the insights of these studies and the findings of the study from my Master’s [53], I designed the Codestrates v2 prototype (Paper D). This prototype was later used in my third case (Paper E). The insights of this case and the insights collected when using Codestrates v1, resulted in the core technological contribution of my Ph.D. project: Varv, a programming model for reprogrammable software as a declarative data structure (Paper F).

This chapter will introduce these four platforms. For each platform, I will briefly summarize its background, provide an overview of its functionality, and discuss the platform in relation to the principles of computational media. For Codestrates v2 and Varv, I will additionally provide information about their implementation process.

### 5.1 Webstrates

Webstrates [162] is a prototypical platform to explore software build as a computational medium. It was published by Klokmose et al. [162] in 2015 and since then matured to a first stable version which was released in 2016. Webstrates is the foundational platform for my research project and the platforms I developed on top of it—Codestrates v2 (Paper D) and Varv (Paper F). I will briefly summarize the platform and its functionality in this section. In the second part of this section, I will relate the design decision of the platform to the principles of computational media.
5.1.1 Webstrates Overview

Webstrates is a platform which runs as a web server. A document in Webstrates is called a webstrate — short for “web substrate.” A webstrate can be accessed using regular web browsers and is served as a website that is identified and addressable by its URL. Contrary to regular websites which only receive the DOM content from a server, the DOM of a webstrate is synchronized with and persisted on the Webstrates server. This means that modifications to the DOM, for instance, using the developer tools of a web browser, are synchronized with the server and with all other clients currently visiting the same webstrate. By storing HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), and JavaScript in the DOM this enables to create collaborative web applications.

Another key functionality of Webstrates is its support for transclusion [226]. Transclusion enables composability of documents in Webstrates by using <iframe> elements to transclude one webstrate into another — similar to how regular websites can be embedded into one another. But not only documents can be transcluded, also parts of documents: For example, a vector graphic that is stored in the DOM of a webstrate in the SVG (Scalable Vector Graphics) format, could be transcluded both into a text document as a figure and into an vector editor webstrate. Changes by the vector editor to the vector graphic webstrate would be directly applied to the embedded vector graphic in the text document.

After its initial publication, a series of APIs (Application Programming Interfaces) [307] were added to Webstrates. These APIs added functionality such as Signaling, which enables to send message to other clients visiting a webstrate without using the DOM, Transients, which excludes <transient> elements or transient attributes from synchronization and persistence with the Webstrates server, and Assets, which stores arbitrary files such as images in a webstrate.

On a technical level, the synchronization of webstrates is achieved by using operational transformation. Any change to the DOM — up to the keystroke level — generates a new operation and with it a new webstrate version. Hence, the versioning is very fine granular and writing applications can quickly result in thousands of versions of a document.

Overall, Webstrates acts as a platform for creating collaborative and malleable software. It was most notably used in the Codestrates v1 [257] and Codestrates v2 [49] platforms as an underlying synchronization and persistence layer. But also apart from Codestrates, Webstrates was used in other projects such as an affinity diagramming application [191] or a digital sticky notes applications [71].
5.1.2 Webstrates as Computational Media

Webstrates sets the foundation for the other platforms presented in this chapter. While it does not offer tools for authoring content, its synchronization and persistence enables to build new authoring tools on top of Webstrates. The following paragraphs summarize how Webstrates realizes some of the principles of computational media.

**Malleability.** Webstrates creates malleable documents, whose implementation is part of the DOM. This makes documents reprogrammable using the developer tools of the web browser. The reprogrammability of Webstrates, however, is limited: Some browsers do not support developer tools, e.g., on mobile devices, and the developer tools are designed for debugging rather than code authoring. Through transclusion it is possible to make documents extensible, as shown in the original Webstrates paper [162], where tools in a toolbar of a webstrate were added by transcluding the code of the tool — following the instrumental interaction paradigm [25]. However, software needs to be designed especially for this kind of extensibility. Webstrates also supports creating copies of webstrates, which enables to use one webstrate as a prototype for copies of it.

**Shareability.** Webstrates implements a variety of facets from the shareability principle. First, webstrates are addressable through URLs. This enables to share a webstrate by simply sharing its URL. Webstrates’ synchronization makes real-time collaboration in a document possible. However, Webstrates only synchronizes the DOM of a webstrate, not the current runtime of JavaScript on its clients. It is, further, not possible to collaborate in real-time when editing documents through the developer tools.

**Distributability.** Documents in Webstrates are distributable across a wide range of devices that support accessing websites in a web browser — desktop computers, laptops, tablets, smartphones, and more. This is possible because Webstrates builds on web technologies and webstrates are accessible in the web browser. Still, Webstrates offers no support to distribute functionality or computation: each client runs their own computation and displays the same functionality in the DOM.

**Computability.** Webstrates supports only limited computability by using the console in the developer tools of a web browser to run JavaScript code. Beyond this, Webstrates does not offer an authoring environment or other ways to execute code in a document by itself. However, Webstrates bundles the computational environment with
documents, as the JavaScript runtime works independently from the client device of a user.

5.2 Codestrates v1

Codestrates v1 [257] is a code authoring and execution platform that was built on top of Webstrates. It was published by Rädle et al. [257] in 2017 as a research prototype. I first used Codestrates v1 in the same year and developed a package manager [52] for it as part of my Master’s project. I also used Codestrates v1 in a study as part of my Master’s [53] and in two of the case studies of my Ph.D. project—the nanoscience lab study (Paper A) and the collaborative writing study (Papers B and C). In this section, I will describe the motivation behind Codestrates v1, briefly summarize the platform and its functionality, and relate design decisions of the platform to the principles of computational media.

5.2.1 Codestrates v1 Background

Codestrates v1 was inspired by computational notebooks such as Jupyter [256] and the literate computing paradigm [248]. Literate computing describes the weaving of executable code and descriptive prose in the same document, for instance, in markup and code cells. Unlike conventional computational notebooks, which were mostly used for data analysis and visualization tasks, Codestrates v1 created a computational notebook for creating interactive application-like software. To make software that was created with Codestrates v1 more extensible and to facilitate reuse of code, I developed Codestrate Packages [52] as part of my Master’s project. This allowed me to deeply engage with Codestrates v1 and Webstrates and motivated me to use the platform also during my Ph.D. project. Through the package management, it was possible to shorten the development time of prototypes—the computational labbook and the malleable word processor—for my first two case studies as functionality could be split into packages. These prototypes focused on extensible software that consisted of tools, bundled as packages, which could be combined in a flexible way.

5.2.2 Codestrates v1 Overview

Codestrates v1 is a platform and authoring environment for Webstrates that runs on the client-side in the web browser of a user (see Figure 5.1). Codestrates v1 structures code in paragraphs—similar to cells in computational notebook like Jupyter. Paragraphs can have one of four types: body paragraphs contain HTML, style paragraphs contain CSS, code paragraphs contain JavaScript, and data paragraphs
contain JSON. These paragraphs are displayed in a one-dimensional linear document, resembling a notebook. Paragraphs could be bundled into sections.

Body paragraphs display their HTML and provide an additional editor to modify it. Style, code, and data paragraphs are displayed as code editors. Changes to the CSS in style paragraphs are immediately applied to the document. Code paragraphs can be either executed manually by clicking on a “run” button or set to “run-on-load,” which would execute them on page load. Codestrates v1 adds its own runtime for executing JavaScript code independent from the normal execution order of the web browser. Developing an application with Codestrates v1, for instance, can be done by adding a body paragraph for the user interface, a style paragraph for styling the interface, and a code paragraph to add interactive behavior to it.

A webstrate that runs Codestrates v1, a codestrate, is self-contained. This means that the implementation code of Codestrates v1 is part of the document itself for each codestrate. Codestrates v1 even goes a step further and adds its own code in the same paragraph types in which user code can be added, making it possible to modify any aspect of the implementation of Codestrates v1 from within the same notebook-like environment.

The package manager [52] of Codestrates v1 can bundle functionality as named packages, which can be shared between multiple codestrates. When installing packages, one codestrate transcludes another, copies over relevant packages from the DOM, and removes the transcluded codestrate again.
Overall, the Codestrates v1 platform enabled it to create collaborative and malleable web apps without the need for external tools. Using the underlying Webstrates platforms as a persistence layer, also no additional server to store data was needed. This made Codestrates v1 an effective prototyping tool and interactive playground, which was used in a variety of contexts: It was, for instance, used in collaborative data visualizations [16, 135, 210] or as a platform for collaborative programming assignments [53].

5.2.3 Codestrates v1 as Computational Media

Codestrates v1 continues the efforts of Webstrates towards realizing the vision of computational media. It, most notably, advances the computability principle of computational media but also improves other principles while inheriting the capabilities of Webstrates. The following paragraphs summarize how Codestrates v1 realizes some of the principles and facets of computational media.

Malleability. Codestrates v1 improves the reprogrammability of documents by adding an authoring environment that is part of the document. It, furthermore, makes it possible to reprogram even the code of the implementation of Codestrates v1 itself, which, in turn, enables to perform even low-level adjustments to how Codestrates v1 works. Codestrate Packages, additionally, enables to create extensible software, consisting of packages that can be added or removed with few clicks and without programming knowledge.

Shareability. Documents in Codestrates v1 are in the same way addressable as in Webstrates. The authoring environment of Codestrates v1, however, enables real-time collaboration when programming as code in paragraphs is synchronized through the Webstrates server. Being able to make changes to documents directly within a document also makes it easier to invite other people into a document to receive help from them, as no additional software is required and Codestrates v1 works independently of the developer tools. Still, the user interface in Codestrates v1, including which paragraphs and sections are toggled visible in a document, is mostly synchronized among all clients. This can cause confusions when trying to work asynchronously.

Distributability. The paragraphs, which are used by Codestrates v1, make it easier to distribute functionality across multiple clients by toggling different body paragraphs into full screen mode on different clients. For instance, in a 3D-model editor, the 3D-model could be edited on the computer. On a second devices such as a smartphone or tablet, the preview could then be displayed in a body paragraph,
which is toggled to full screen, enabling to use touch gestures to rotate or resize the model preview. This was, for instance, possible in the computational labbook prototype (Paper A).

**Computability.** Codestrates v1 improved the computability of the platform in multiple ways. First, it added an authoring environment to Webstrates. This allowed to author code without the need for external tools such as the developer tools. Next, it added a runtime for code execution, which enabled users to run code directly within the document and at any time—not merely when the `<script>` element in a document is usually loaded on page load. This was inspired by computational notebooks, which similarly allow users to run cells individually.

5.3 Codestrates v2

Codestrates v2 [49] is a generalization of the Codestrates v1 platform and adds flexible development platform to Webstrates. It splits the platform up and consists of three components: The Webstrates Package Manager, the Execution Engine, and the authoring environment Cauldron. Parts of this section are based on the technical report of Codestrates v2: Paper D (see Chapter 12).

5.3.1 Codestrates v2 Background

Codestrates v2 was motivated by the desire to create a more modular and flexible development platform for Webstrates. While Codestrates v1 allowed to develop applications in Webstrates from within the web browser without the need for the developer tools or other external tools, it felt restrictive with regards to its user interface, which was motivated by computational notebooks.

In Codestrates v1, the execution engine of code paragraphs was tightly coupled with the overall notebook structure of documents, and the notebook user interface. This, for example, was restrictive when users wanted to view multiple paragraphs side by side at the same time—as my work during my Master’s [53] demonstrated. Furthermore, it slowed down loading times of a codestrate when an user would merely want to use but not modify it, because the user interface was always loaded as well. Lastly, we found that the literate computing paradigm, as it was implemented in Codestrates v1, caused confusion and breakdowns for its users. Placing code in paragraphs in the same space of a document as the tools or application a user is working with, blurred the border between using and developing software—an initial goal of Codestrates v1 [257]. This style of development, however, was in contrast with what users with programming experience were familiar with. For instance, in the second case—Papers B and C—
participants struggled to distinguish which part of a document was part of the interface from the actual prototype and which came from Codestrates v1 user interface. In Codestrates v2, we aimed to overcome these challenges by creating a modular platform and relying on a more conventional authoring environment.

5.3.2 Codestrates v2 Overview

Codestrates v2 is a development platform for creating collaborative web applications with Webstrates. It generalizes the key features of Codestrates v1 and provides a more general purpose platform, rather than a prototype focused on literate computing and the notebook structure. Overall, Codestrates v2 consists of three components that work together to create the development platform: the Webstrates Package Manager, the Execution Engine, and the authoring environment Cauldron (see Figure 5.2). This section will provide a brief overview of the three components and their functionality. For a comprehensive description of the platform I refer the reader to Paper D, which is included as Chapter 12.

WEBSTRATES PACKAGE MANAGER. WPM (Webstrates Package Manager) is a continuation of the package manager of Codestrates v1. WPM makes it possible to import and reuse code or functionality as packages from other webstrates. In contrast to the package manager of...
Codestrates v1, which only allowed to bundle whole sections as a package, WPM makes it possible to bundle any element in the DOM as a package. Besides DOM elements, packages contain meta data, e.g., a description and version, and assets like images or code libraries.

Packages can be installed by requiring them: One webstrate serves the packages and another requires the package from it. Packages can be either installed transiently, i.e. copied into a transient element in the head of a webstrate during each page load, or persistently, i.e. copied into the persisted part of the DOM of a webstrate.

**Execution Engine.** The execution engine handles the execution of code in Codestrates v2. The “computational units” of Codestrates v2 are called code fragments. Code fragments can contain code from various types of programming languages (JavaScript, TypeScript, Python, Ruby, and Lua), markup languages (HTML, Markdown, LaTeX), and style sheet languages (CSS and SCSS (Sassy Cascading Style Sheets)). The code in fragments is stored as plain text in the DOM and, thus, is not interpreted by the web browser by default.

Fragments that contain executable code, can be run individually or set to auto-run on load of a webstrate — similar to code paragraphs in Codestrates v1. It is, furthermore, possible to use different programming languages inside the same codestrate. Functions from fragments with one programming language can be exported and imported into fragments with another programming language. Hence, Codestrates v2 is a polyglot system. Code from markup and style sheet languages can only be toggled to auto-run but not be run manually. Toggling such a fragment to auto-run, processes the code, creates a transient view element adjacent to the fragment, and renders the processed code into it. For instance, Markdown is rendered into an HTML element and CSS is placed in a <style> element.

The execution engine, lastly, provides functionality to create code editors and views in the DOM. Code editors make it possible to edit their code directly in the web browser and, because the content of fragments is persisted, enable collaborative editing of code. Views enable to render content from, for instance, an HTML fragment in multiple locations in the DOM — all automatically receiving updates from the fragment. In contrast to Codestrates v1, which only created one view and one editor for each paragraph, Codestrates v2 can create multiple views or editors that are synchronized with the same code fragment.

**Cauldron.** Cauldron is Codestrates v2’s authoring environment (see Figure 5.2). It comes with a user interface including a tree browser, inspector, and tabbed editor. While the tree browser enables users to browse the DOM elements and fragments of a webstrate, the inspector can be used to edit attributes of these elements, e.g., the id or class. The tabbed editor uses the execution engine to create code editors and views in the DOM.
places them in a tabbed authoring environment. This enables, similar to a regular code editor or IDE, to open multiple fragments in tabs and view them, for instance, side-by-side.

We decoupled the execution engine from the authoring environment to make the platform more flexible: It makes it possible to author the same code using different authoring environments and cuts the loading times of the platform as Cauldron is not loaded by default but is only loaded once the “Edit” button in the corner of a webstrate is clicked.

Overall, Codestrates v2 provided a more robust and flexible development platform for Webstrates than Codestrates v1. Beyond that, it added APIs to create editors and views of fragments in multiple locations inside a document. The improved performance compared to Codestrates v1 also made it more feasible to create more computationally heavy applications, such as the malleable, hybrid video conferencing platform MirrorBlender [118].

5.3.3 Codestrates v2 Implementation Process

Codestrates v2 was implemented by the two programmers Rolf Bagge and Janus Bager Kristensen from CAVI [58], a research center at Aarhus University. At the time of development, Rolf and Janus already maintained the code base of Webstrates and had experience in implementing software with Webstrates, e.g., in Videostrates [164]. This helped to drastically reduce the development time required to implement the individual components of Codestrates v2. While the implementation was performed by Rolf and Janus, I defined the design specification of the system, supervised, and tested the implemented components.

In order for the programmers to work with their familiar development environment, Codestrates v2 was implemented with external tools. This stood in contrast to Codestrates v1, which was bootstrapped and developed from within itself—even the system components of Codestrates v1 were implemented using the same paragraph types users had available for the implementation of applications. This trade-off between following principles of a vision and realizing a working experimental prototype is also discussed in Paper E.

5.3.4 Codestrates v2 as Computational Media

The design and development of Codestrates v2 not only yielded a new platform, but I also learned about the challenges of creating software that follows the principles of computational media. In the following paragraphs I will summarize the how Codestrates v2 implements some of the principles and facets of computational media.
**Malleability.** Codestrates v2 continues the effort of the Codestrates v1 platform to create an extensible and reprogrammable development platform for Webstrates. It improves the extensibility of Codestrates v1 by reintroducing the WPM as a more generalized and flexible version of Codestrates v1’s package manager. Reprogrammability is still available by granting users access to code from within the document — merely a press on the “Edit” button away.

While the malleability of applications implemented with Codestrates v2 is similar to the one of applications created in Codestrates v1, the malleability of the development platform itself is different: Codestrates v1 was implemented using the same paragraphs as applications implemented with Codestrates v1. In Codestrates v2, however, due to the different implementation process of using external software, the platform itself, e.g., the execution engine and Cauldron, are not as easily modifiable as in Codestrates v1.

**Shareability.** In contrast to the computational notebook user interface from Codestrates v1, which was synchronized across clients, Codestrates v2 enables more asynchronous ways of collaboration by not synchronizing which Cauldron editors are currently displayed in the tabbed editor across different clients. This makes it easier for collaborators to work together in different parts of the same document. Codestrates v2 also makes it possible edit HTML fragments in real-time collaboratively — a feature Codestrates v1 was lacking.

**Distributability.** The distributability of documents in Codestrates v2 is similar to the one of Codestrates v1 and Webstrates. Functionality, however, is not that easy distributable anymore, as fragments, unlike paragraphs in Codestrates v1, cannot be individually toggled full screen anymore. This pushes the effort to developers that need to create their own transient user interfaces for different clients.

**Computability.** Similar to Codestrates v1, Codestrates v2 supports code authoring and execution directly in the document. However, it improves on polyglot programming capabilities: While Codestrates v1 by itself only supported JavaScript as a programming language, Codestrates v2 expands this with TypeScript, Python, Ruby, and Lua. Codestrates v2 goes another step further and supports code from these fragments to be exported and imported into each other — even across different programming languages.

5.4 **VARV**

VARV [51] is a programming model that was designed for the use in malleable software such as computational media. It represents reprogrammable interactive software as a declarative data structure. This
helps to overcome some of the existing issues with the JavaScript runtime, which was used in Webstrates and both iterations of Codestrates so far. Parts of this section are based on Paper F (see Chapter 14).

5.4.1 Varv Background

Varv was motivated by the need for a more fitting programming model for computational media: a programming model which puts emphasis on malleability, shareability, and distributability. The programming model should make it possible to re-run code after editing it at runtime, to collaborate on reprogramming activities, and to distribute state and interactive behavior across multiple clients.

Previously, in Webstrates, Codestrates v1, and Codestrates v2, we mostly used the imperative JavaScript programming model for implementing interactive behavior — the native programming model of modern web browsers. Using the native programming model enabled compatibility with web browsers and opened up Webstrates to a broad range of existing JavaScript libraries and allowed users to make use of prior knowledge of programming for the web. JavaScript, however, was not designed for editing its code after execution and re-executing it. In two of my case studies, Papers C and E, we identified issues related to using this programming model, which limited the Codestrates v1 and Codestrates v2 platforms in regards to reprogramming code and collaborating in real-time on code. For instance, in Paper C, participants had difficulties understanding when code is executed and how to run code “on load.” In Paper E, participants struggled with issues like duplicated event listeners in their games due to re-running the same code and thereby adding new event listeners on top of the existing ones. Similarly, when collaborating on programming their games, participants had issues, where one group member was running different versions of the code than their team mates, causing errors on the other clients, or running unfinished code that was currently edited by a group member. The issue of an unfitting programming model was also raised by Tchernavskij et al. [288], who called for “better programming models for shareable dynamic media.”

A related term to these problems is divergence: Basman et al. [21] describe divergence as the differences between the runtime state of an application and the state of the code that can be externally authored by users. For instance, changing the code of a JavaScript fragment in Codestrates v2 only changes the plain text of the fragment, to apply the changes the webstrate either has to be refreshed or the fragment needs to be run manually. The latter case, however, created problems like adding another event listener with the new code to a button but not cleaning up the old event listener. Changing code should more directly influence the runtime of an application, hence, reducing the divergence between the two.
To overcome these issues, we set out to create a new programming model that enables this live editing of interactive behavior, without causing divergence or requiring a refresh of the page to apply changes. We took motivation from declarative programming where the code users are writing defines the *what* of the behavior, not the *how*. The latter is then figured out by the system itself. We also took inspiration from trigger-action-based systems like IFTTT [136] (If This Then That) and functional reactive programming systems, such as RxJS [268]: Instead of adding event listeners manually in the code, users should be able to just define the interactive behavior and the system takes care of the low-level changes.

We envisioned that users should be able to make modifications like changing existing behavior of an application or adding new behavior. For example, in a todo list application implemented in Varv, it should be possible to modify existing behavior like how todo items are represented, to extend existing behavior like adding an “assignee” field to a todo item, or to add completely new behavior like a filtering functionality or search for todo items. All of these should be possible at runtime and also in collaboration with other users. Our motivation in the long term is to enable also end-users without programming knowledge to accomplish such modifications. For Varv, however, we focused on users with programming knowledge—we see Varv as a first step towards this goal and as laying a foundation towards declarative programming of interactive behavior.

5.4 Varv Overview

Varv is a programming model for reprogrammable interactive software. In Varv, interactive behavior of applications is defined as a declarative data structure. Varv uses a *concept language*, which is inherently extensible and Varv’s architecture decouples the abstract application logic from its concrete interaction modality. This section will provide a brief overview of the design goals, the concept language, the architecture of the system, and a summary of how Varv reflects some of the principles of computational media. For a comprehensive description of the platform I refer the reader to Paper F, which is included as Chapter 14.

**Design Goals.** The design of Varv was motivated by three design goals. The first goal was to provide a structured and declarative representation of interactive software. We did this by using JSON as the underlying data structure of Varv’s concept language. Through a declarative representation users should be able to focus on the *what* of the interactive behavior while Varv figures out the *how*. In doing so, we also aimed to make reasoning about the semantics of interactive software easier. Using a structured representation, further,
made it possible to hot swap the declarative specifications to enable live (re)programming.

The second goal focused on supporting accretive extensibility. Instead of explicitly designing software to be extensible by, for example, by creating APIs, applications created with Varv should be inherently extensible. Varv should allow this also through mere addition of code and without having to change the existing code. This would, further, facilitate experimentation, where users can add and test new code through addition but also revert back to the old code by just disabling the new code.

The third goal, finally, aimed at decoupling the abstract application logic, e.g., removing an item from a todo list, from the concrete interaction modality, e.g., clicking on a trashcan symbol on the item using the mouse. This makes it possible to reuse the same application logic across multiple modalities and enables users to create semantically-meaningful abstractions for events, for instance, pieceSelected in a board game instead of click. These abstractions, in turn, could be used to create a domain-specific language for a specific application type.

CONCEPT LANGUAGE. The concept language of Varv is used to define interactive behavior of an application (see the accompanying video\(^1\)). The concept language is written in JSON (see Figure 5.3). The core building blocks of the language are concepts: individually named units of interactive behavior. A concept consists of two main parts: a schema and actions. The schema defines the shape and type of data associated with a concept through named properties. Properties can be either of a primitive data type, e.g., string or number, or a reference to another concept. For example, a todo list concept can contain a property that is an array of todo items. Actions of a concept are named and provide an abstraction for state transformation, for example, through interaction or reaction on other changes to the state.

Varv is an reactive and event-based system — all changes to state are performed using events. Events are emitted by triggers, for instance, a click trigger, which emits an event whenever the user clicks. An action defines a when-block and a then-block to react to events. The when-block defines an array of trigger an action is listening to, while the then-block defines an array of actions that should be executed once the trigger emits an event. This array of actions is executed as a chain: each action receives the event, performs its computation, and passes the possibly altered event to the next action in the chain. Varv provides a set of primitive actions, which can be used in the then-block. Beyond that, users can also define their own named actions in concepts and use those in other actions — or within the same action, creating a recursive action.

---

\(^1\) Accompanying video: [https://youtu.be/XCW-LXFBqXE](https://youtu.be/XCW-LXFBqXE) (Retrieved February 22, 2022)
Figure 5.3: Varv concept language example. It shows the concept definition of a todo list applications. The example is described in detail in Paper F in Section 14.2.2. (Quotation marks from the JSON keys removed for readability.) (Reprinted from Paper F)

Users can create multiple concept definition files, which are merged by Varv into one concept definition. If keys exist in multiple definitions, the ones further down in the document overwrite earlier ones. This makes it possible to modify an application through addition by merely overwriting the parts that need to be changed while leaving the rest untouched. Additionally, the concept language adds four operators for extending concepts, which, for instance, allow users to join concepts together or inject one into another.

**Architecture.** The architecture of Varv (see Figure 5.4) is split up into six main components: the event engine, concept definitions, data stores, mappings, templates, and views. Concept definitions are files that use the concept language. They are parsed, merged, and applied as the model by the event engine. Whenever they are changed, the event engine rebuilds the whole model and hot swaps the old one with the new one.
Views and Templates

Views can connect to the event engine to render a concrete representation of the application and its state. Multiple views can be connected to the same event engine at the same time. Templates are used to specify how state should be represented in a view. Templates are view-dependent, for instance, the DOM view uses HTML as the language for the templates. To emit events to the event engine, views can also define view-dependent view triggers.

Data Stores and Mappings

Data stores in Varv enable to store the state of concepts and their properties. Varv stores data in a simple key-value format. Thus, the only requirement for a data store is to support storing this kind of data. Similar to how multiple view can connect to the event engine, also multiple data stores can be used with the event engine. Mappings are pointers that define which property of which concept is stored in which data store. The event engine forwards state changes to the data stores, while data stores can also send updates to the event engine. Sending updates to the event engine is useful when, for example, the data store is shared and used by multiple clients. In our Webstrates implementation of Varv, data is by default stored in the DOM and synchronized by the Webstrates server. Thus, changes of other clients are forwarded to the event engine in this way.

5.4.3 Varv Implementation Process

Varv was designed and developed over the course of over one year. During this time, we iterated over two main versions of Varv: The first prototype, which was called WhenJS during its development, did not have the structured nature of Varv but instead focused on just consisting of a set of when-then rules. WhenJS also did not separate the view and the data store, storing both in the same space directly.
in the DOM. Once the first prototype was functional, we explored developing various applications using it. This, however, uncovered multiple problems in our specification and we re-iterated over the general structure and syntax of the system.

This second iteration resulted in a comprehensive design specification of a first version of the concept language used in the final version of Varv. Again, once the prototype was functional, we re-implemented the WhenJS examples in Varv to assess the expressivity of the concept language. In this second iteration, we also introduced the idea of templates and data stores to decouple the event engine from the view and the data store.

The development process was, like Codestrates v2, supported by Rolf Bagge and Janus Bager Kristensen. While I defined the design specification of the language, Rolf and Janus worked on the technical implementation of the system. Provided with snapshots of prototypes, I implemented examples and iterated over the design specification.

5.4.4 Varv as Computational Media

Varv sets its main focus on creating a programming model that matches the values of computational media. In doing so, it advances not only the computability principle but also the other three principles. While Varv can also be used without Webstrates, e.g., we created a proof-of-concept Electron implementation, it is synergetic when bundled with Webstrates and Codestrates v2 as its underlying platform.

**Malleability.** Varv’s programming model inherently supports reprogramming and extension. By representing interactive behavior in a structured form and by enabling the incremental extension of this behavior, Varv makes it possible to extend or reprogram any aspect of an application through addition. Due to its decoupled data stores and views, Varv, furthermore, makes it possible to create personalized views of applications, while using the same underlying data store, which improve its interoperability. Still, the current implementation of Varv requires users with programming knowledge to implement applications. In future work, we plan to reduce this threshold to also enable users with only little or no programming skills to make use of Varv’s accretive extensibility.

**Shareability.** As Varv can apply changes to concept definition files directly while changing them, the interactive behavior in runtime always represents the code in the definition files. Hence, by synchronizing the concept definitions through the DOM in Webstrates also the runtime state is synchronized across multiple clients. This applies to all properties that are stored in the DOM. Properties that are stored purely in memory are not synchronized and, hence, enable to retain a
creating computational media

relaxed WYSIWIS (What You See Is What I See) model. Because of this synchronized runtime state, Varv allows for real-time collaboration on programming and testing applications.

**Distributability.** When used together with Webstrates, Varv retains the possibility to distribute documents easily. Also the distribution of functionality is similar to the one available in Codestrates v2. However, with Varv users can create personalized views while using the same underlying data. This makes it, for instance, possible to create create views for certain aspects of an application: in a todo list one document could display the list of todos while in another document the input field to add new items is displayed. By using the same data store in both, state would be synchronized between both documents.

**Computability.** Besides malleability, computability is the main principle Varv addresses. By offering a programming model that better matches computational media, it makes it easier to modify existing code and collaborate on code in real-time. Additionally, Varv goes a first step towards addressable user interfaces: Using the DOM view, Varv generates user interfaces using a template and state of a data store. Thus, it enables to connect elements in the user interface to their underlying data and makes them addressable in the concept definitions.

5.5 Summary

In this chapter, I investigated the second research question of how to create concrete prototypes that follow the vision and principles of computational media. First, I revisited the Webstrates and Codestrates v1 platforms, which already enabled some of the principles and facets. Second, I presented two platforms I have created during my Ph.D. project: Codestrates v2 and Varv. I presented an overview of all four platforms and related each of them to the four principles of computational media. Table 5.1 shows an overview of which facets each of the platforms supports. In the next chapter, I will present how I used these platforms in three case studies throughout my Ph.D. project.
Table 5.1: Overview table of how the facets of principles are implemented in the four platforms presented in this section. An × indicates that the facet is fulfilled, an (×) indicates that the facet is partially fulfilled.
In this chapter, I present insights from using computational media prototypes and platforms based on Webstrates and Codestrates in three case studies I conducted during my Ph.D. project (Papers A, B, C, and E). I will summarize findings from these empirical cases. These findings relate to the third research question about what potentials and challenges computational media can bring for its users. I chose the Webstrates and Codestrates platforms as the technical foundation for the prototypes used in these cases based on two reason: On the one hand, Webstrates and Codestrates embody many of the principles of computational media and allowed me to investigate different aspects of computational media at the same time. On the other hand, Webstrates and Codestrates were a pragmatic choice: Having had experience in using both platforms prior to my Ph.D. project allowed me to speed up the development process of prototypes.

The choice of focusing on the particular cases presented in Papers A, B, C, and E evolved organically during my Ph.D. project—I discussed this in the project timeline in Section 3.1. These cases allowed me to explore facets of computational media in different context. The cases and papers, however, also present results and findings that go beyond my Ph.D. project, e.g., computational characteristics of a nanoscience lab or a trade-off protocol for collaborative writing tools.

6.1 CASE 1: NANOSCIENCE LAB STUDY

In this case, my co-authors and I collaborated with a nanoscience lab in Aarhus to create a computational alternative to turn-key applications and homebrew scripts: the computational labbook. Parts of this section are based on Paper A (see Chapter 9).

6.1.1 Case Background

This case was motivated by ongoing research of my co-authors in this project, Midas Nouwens and Clemens Klokmose, who conducted observations and interviews with researchers of a nanoscience lab at Aarhus University. Through their observations, they found that the researchers’ current use of applications and scripts to use computation in their scientific work faces various challenges: The researchers operate on the cutting edge of science without established tools for their work. Therefore, they rely on applications made by other labs that work in similar fields or custom scripts that a programmer in
their lab is creating just for them. This causes them to having to switch between a variety of applications, command line scripts, code editors, and text documents or reports for their work. After these observations and interviews were conducted, I joined the project.

These preliminary findings prompted the idea of exploring how a computational medium could overcome some of these challenges. To explore this idea, we set out to conduct a participatory design process together with the researchers of the nanoscience lab. An initial goal, which we later in the process abandoned, was to combine the functionality of the scripts, applications, and text documentation in a single computational labbook.

6.1.2 Case Overview

Three Activities

This case study consisted of three main activities: observations and interviews, a participatory design process of a prototype, and in-situ interviews of participants using the prototype. The following paragraphs will briefly summarize the conducted research I participated in: the participatory design process and the in-situ interviews. For a detailed description I refer the reader to Paper A, which is included as Chapter 9.

Collaboration With Nanoscience Lab

Participants. We collaborated with a biomolecular nanoscience lab at Aarhus University. Participants of the study were researchers of this lab, which consisted of 13 researchers: one principle investigator, eight postdoctoral researchers, and four Ph.D. students. As the activities of this project span over two years, not all participants participated in all activities. The researchers in this group were not trained as programmers and only had little to no programming skills.

Establishing a Possible Future

Participatory Design Process. To explore how a computational medium could support the researchers in their work, we aimed to establish a possible future [269] of software that is not based on application-centric software but on a malleable computational medium. We used the development of the prototype to co-create a future vision of computational labbook (see the accompanying video\(^1\)), i.e. a computational laboratory notebook, together with the participants. We wanted participants to reflect on their current use of applications and scripts and how this could be improved. Through a series of design meetings and interviews, we created a total of four iterations of a prototype together with the participants. The prototype was implemented by me.

Using the Prototype

In-situ Interviews While Using the Prototype. The fourth and last iteration of the computational labbook prototype (see Figure 6.1) was used in three in-situ interviews with three researchers of

\(^1\) Accompanying video: https://youtu.be/hoe5YnqsAAM (Retrieved February 22, 2022)
6.1 CASE 1: NANOsciENCE LAB STUDY

Figure 6.1: Screenshot of the computational labbook prototype. The center shows the sections and paragraphs of the document. On the right side the Instrument Panel allows users to drag instruments into the document. (Reprinted from Paper A)

the lab. The interviews were semi-structured and started with an introduction to the prototype, its features, and an explanation of the four principles of computational media. After the introduction participants were asked to use the prototype to design a molecule with it—the same task they currently did using applications and scripts. Based on the prototype, we started a discussion with participants on their use of computation in their work and how the principles of computational media and computational media in general relate to their work. The interview was lead by my co-author Midas, while I presented the prototype to the participants.

DATA COLLECTION AND ANALYSIS. We recorded the audio of each of the three in-situ interviews, which lasted 82 minutes, 57 minutes, and 90 minutes. Me and my co-authors each coded the interviews independently using open coding. We focused on statements that were related to any of the four principles of computational media. In three joint meetings, we collected all codes, noted them on a whiteboard, and discussed and arranged them into themes.

OTHER OUTCOMES. Besides the perspectives on computational media, which I will discuss in the next subsection, this paper also contains other contributions: First, we presented an overview of the nanoscience lab and their work environment. Second, we described three computational characteristics that define how participants structured and shaped their work and their use of computational tools in their work. These contributions are included in Paper A in Chapter 9.
6.1.3 Perspectives on Computational Media

In investigating the use of computation in the nanoscience lab, we learned several things about computational media and its principles. Overall, we quickly found out that our initial goal of consolidating the different applications and scripts into a single computational labbook was neither feasible nor desirable. Instead, we found that computational media can act as a third type of medium between applications and scripts for the researchers: While applications are often stable and user-friendly, they are also rigid and do not allow for custom changes that are required in the quickly evolving field in which the nanoscience lab works in. Scripts, on the other hand, allow the researchers to create highly custom scripts for their specific use cases, albeit at the cost of stability and user-friendliness. Computational media is placed in between the two: While it is customizable and malleable, it does not sacrifice all stability and user-friendliness. Our computational labbook made it possible to create small tools that can be based on custom scripts, but also provide a user interface and exception handling to make them more accessible for the researchers with only little programming skills.

In the following paragraphs, I will summarize findings for each of the principles and their facets. Table 6.1 provides a summary of these findings.

<table>
<thead>
<tr>
<th>PRINCIPLE</th>
<th>POTENTIALS AND CHALLENGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malleability</td>
<td>P: Enables changing functionality through more capable peers.</td>
</tr>
<tr>
<td></td>
<td>C: Concerns about malleable data entries and authenticity of data.</td>
</tr>
<tr>
<td>Shareability</td>
<td>P: Receiving help from a more capable peer.</td>
</tr>
<tr>
<td></td>
<td>C: Concerns about data provenance.</td>
</tr>
<tr>
<td>Distributability</td>
<td>P: Offloading heavy computations.</td>
</tr>
<tr>
<td></td>
<td>C: Requirement of a central server.</td>
</tr>
<tr>
<td>Computability</td>
<td>P: Polyglot programming allows for opportunistic use of scripts.</td>
</tr>
<tr>
<td></td>
<td>P: No version incompatibilities of scripts.</td>
</tr>
<tr>
<td></td>
<td>C: Moving the computational environment out of the control of users.</td>
</tr>
</tbody>
</table>

Table 6.1: Summary of the potentials and challenges, sorted for each of the four principles of computational media, found in the first case, the nanoscience lab study. Potentials are denoted with the letter P and challenges with the letter C.
Malleability. We found that malleability was not a direct requirement for the researchers that use the scripts. However, it was essential when working together with a more capable peer: By being able to change the implementation code of tools in the computational labbook right within the computational labbook, it would allow researchers to ask the programmer in their group for help in creating and modifying scripts of the tools they use. A central mechanism to facilitate this was the package management of Codestrates v1, which would allow the programmer to create new tools based on scrips and publish them in a repository. The researchers could then install these tools into their documents and use them right away.

However, there were also concerns from the researchers regarding the persistence of scientific records. By making tools and the computational labbook malleable, data entries and results could be changed after the fact. An ability to “freeze” certain parts of a document so that results could be authentically replicated would be necessary.

Shareability. In the case of the researchers, we found that shareability was not essential for their workflow as they often work individually on projects and molecule designs. They had no immediate need for real-time collaboration. However, when it came to receiving help from colleagues or a more capable peer, it was essential to have the document in a common space, which would allow them to easily share a link to the document with a colleague. This is especially useful due to the embedded computational environment (see below in Computability).

Regarding the use of computation in a shared document, we saw that the researchers raised concerns about scientific provenance of data: By having a shared computational document, inevitably, questions about “Is something computing right now?” or “Who performed this computation?” would come up in the future. These questions relate to other research on groupware, where questions like “Who did what?” can come up when multiple people access the same document [122].

Distributability. Distributability had varying importance for the researchers: Distribution of documents was important for them, as they moved between the lab space and the office space and needed to access documents in both places. While they achieved this goal already partially by using file-sharing platforms and cloud services, these services would only share the files but not the applications to open them — these applications would still need to be installed and maintained on all their devices.

Distributing functionality, e.g., by opening an 3D-viewer of a molecule on the tablet and the design on the laptop, was not important to the researchers. Even though they had both laptops and tablets, they preferred to only use their laptop for this kind of work.
This might relate to a legacy bias, where users only use one or two devices even if more are available [254].

Distribution of computation, however, was very useful to the researchers: As some of their scripts require heavy computational power, the scripts often stalled the computationally weak laptop of the researchers, causing them to wait for hours until the laptop was usable again. By offloading such computations to a powerful server, computation times could be cut down significantly and laptops would remained functional and could even be turned off while the computation was running on the server. A related concern of this model was the central server required in our prototype as the researchers would have valued being able to also work offline on their laptops. Decentralizing the prototype could have helped to resolve this issue.

**Computability.** As the field of the researchers lacks a set of established applications they can use for their work, they have to opportunistically use scripts from other research labs or create their own scripts. Over time, this causes the scripts they use to be written in different programming languages or even different versions of the same programming language. The polyglot nature of the computational labbook allows scripts of different programming languages to be used and executed on a central server — an important feature for the kind of experimental work of the researchers.

Another benefit we found was that the computational environment was embedded in the document. This meant that scripts run with the same underlying computational environment (e.g., libraries or programming language versions) and incompatibilities with certain devices types or operating systems could be avoided. This was a source of errors for the researchers prior to our prototype, as some scripts required certain versions of Python or Perl distributions to run.

### 6.2 Case 2: Collaborative Writing Study

The second case I am reporting on focused on the use of computational media in prototyping workshops as part of a co-design study on collaborative writing. For these workshops I created the *malleable word processor* prototype. Parts of this section are based on Papers B and C (see Chapter 10 and Chapter 11).

#### 6.2.1 Case Background

This case aimed at exploring how computational media can support participatory design or co-design processes and cooperative prototyping. This was partially motivated by employing participatory design in my first case study. While in the first case study I implemented the ideas of participants, in this case study, I aimed to explore to what de-
gree participants themselves can make use of malleable software when co-creating a prototype. To do this, I teamed up with my co-author Ida Larsen-Ledet to facilitate a co-design study, which aimed to explore the idiosyncratic practices of co-writers when collaboratively writing texts.

The initial goal of this project was to conduct a three-stage co-design study, in which we would prepare a malleable word processor prototype for the last stage. This prototype would then be used and modified by participants to customize their word processor to their own needs and to implement ideas for tools they came up with in earlier stages of the study.

6.2.2 Case Overview

We conducted a co-design study around the domain of collaborative writing in academia. The study consisted of three stages and a total of five workshops. The following paragraphs will briefly summarize the conducted research. I will put an emphasis on the third stage, because this is the stage we used computational media in form of the prototype. For a detailed description I refer the reader to Paper B, which is included as Chapter 10 and details the overall methodology of the co-design study and focuses on the collaborative writing aspect of the project, and to Paper C, which is included as Chapter 11 and focuses on the prototyping workshops in the last stage, using the malleable word processor prototype.

The co-design study consisted of three stages and five workshops — each workshop lasting two hours. The first stage consisted of two identically structured workshops in which we identified problem spaces in collaborative writing through various creativity techniques. The outcome of this stage were problem domains

Participants. Overall, 18 people participated in the co-design study. The participants were recruited with a focus on experience with collaborative academic writing, for instance, in writing scientific papers or a Master’s thesis. As the co-design study took place in person, we recruited participants from our own university. Participants’ occupation ranged from Master’s students, Ph.D. students, postdoctoral researchers to assistant or associate professors, and also included recent graduates working as software engineers. Most participants had a technical background and were working in technology- or design-related fields. Participants could participate in any number of stages of the co-design study, depending on their availability. In the prototyping workshops, a total of 13 people participated. All participants had some programming knowledge, yet, the frequency of programming ranged from daily to rare use.

Academic Writers

The co-design study consisted of three stages and five workshops — each workshop lasting two hours. The first stage consisted of two identically structured workshops in which we identified problem spaces in collaborative writing through various creativity techniques. The outcome of this stage were problem domains
Using Computational Media

Introduction

Motivation

I want to share data on screen, ensure everyone is on the same page, and keep everyone at the same screen. The ability to look at the code and view changes is essential. People tend to use their own tools and software, which can lead to misunderstandings or delays. The challenge is to create a shared understanding and keep everyone on the same page.

Overview

Researchers have developed new tools for collaborative writing and initial ideas for tools that could resolve some of these problems.

The second stage consisted of a single workshop in which we generated ideas about how the problems, which we identified in the first stage, could be overcome. We used the initial ideas from the first workshops to start discussions. The outcome of the second stage were more detailed and elaborate ideas for tools. The first two stages were mainly planned by my co-author Ida.

Based on the ideas from participants from the second stage, I developed a malleable word processor prototype in between the second and third stage of the study. The prototype consisted of a rich-text word processor and an array of tools that were based on participants’ ideas (see the accompanying video2). The prototype was built on top of Codestrates v1 and tools were deployed as packages using Codestrates Packages. The prototype had two views: a writing view, which hid most of the controls of Codestrates v1 (see Figure 6.2a) and the regular view of Codestrates v1 (see Figure 6.2b).

In the third stage, which like the first stage consisted of two identical workshops, we then let users use and modify the prototype. Participants worked in pairs or groups of three to use the prototype and modify it using the development environment within Codestrates v1. Participants used their own laptops for this task. While modifying the prototypes, participants were also asked to fill out storyboards of ideas they could not realize due to missing time or lacking programming skills, and a free-text questionnaire about the prototyping workshop and malleable software in general. The outcome of the third stage, finally, were modified tools of participants, storyboards, and question-

Figure 6.2: Screenshots of the prototype’s two views. The views can be switched with the “Switch Theme” menu entry on the left. (Reprinted from Paper C)

2 Accompanying video: https://youtu.be/aLAvVnRzUg (Retrieved February 22, 2022)
naire answers. In the end of the workshops of the third stage, each group briefly presented their results, followed by a plenary discussion.

**DATA COLLECTION AND ANALYSIS.** In the beginning of the co-design study, we collected anonymized demographic questionnaires. During the study, we recorded audio and video, and took photos. In each stage, we additionally collected the artifacts the participants created, for instance, notes, diagrams or cards with their ideas. In the third stage, we collected the modified prototypes of the groups of participants. Furthermore, we recorded the audio and video of each group while they used and modified the prototype using portable action cameras placed behind the participants.

We analyzed the data of the third stage, the prototyping workshops, using a qualitative analysis with the recordings as our primary data source. We categorized incidents in the group work using labels like “discussion” or “problem.” The plenary discussions were transcribed and, together with questionnaire data, further informed our analysis.

**OTHER OUTCOMES.** The papers of this project also include other contributions: First, we identified themes of challenges participants were facing when writing collaboratively in an academic environment. Second, we discussed contrasting needs of different participants to their collaborative writing tools and practices. We presented a proposal of a trade-off protocol to aid addressing such contrasting needs both when designing and evaluating systems. Lastly, we describe lessons learned on how to use a reprogrammable prototype in cooperative prototyping workshops with limited resources, e.g., limited time with participants. These contributions are included in Papers B and C in Chapter 10 and Chapter 11.

### 6.2.3 Perspectives on Computational Media

Using computational media as a tool for prototyping as part of this case helped us to identify a variety of shortcomings of computational media for collaborative prototyping and programming. Additionally, it helped us to gain insights into the diverse and often contrasting needs of users. These contrasting needs, further, motivate my research in the direction of malleable software. Additionally, we learned that the programming model of literate computing, which is used in Codestrates v1, can hamper reprogramming activities rather than supporting them. As most participants were unfamiliar with this type of programming, it required them to rethink their programming practice for this new medium.

In the following paragraphs, I will summarize findings for each of the principles and facets of computational media. Table 6.2 provides a summary of these findings.
### Table 6.2: Summary of the potentials and challenges, sorted for each of the four principles of computational media, found in the second case, the collaborative writing study. Potentials are denoted with the letter P and challenges with the letter C.

<table>
<thead>
<tr>
<th>Principle</th>
<th>Potentials and Challenges</th>
</tr>
</thead>
</table>
| **Malleability** | P: Even in a short time frame, people can make meaningful changes to software, using reprogrammability.  
|             | P: Extension of functionality through a package management is an easy way to customize software.  
|             | P: Contrasting needs in collaborative software can benefit from malleability in order to resolve conflicts and users having their own views on a document.  
|             | C: Reprogrammability needs more guidance to be useful to the user.  |
| **Shareability** | P: Enabled participants and facilitators to collaboratively edit the prototype.  
|             | C: Users can easily disturb each other when working in the same document at the same time.  |
| **Distributability** | P: Participants of workshops can bring and use their own device.  |
| **Computability** | P: Participants are not required to install any software in order to use and modify the prototype.  
|             | C: Applying changes through re-running of code caused erroneous documents.  
|             | C: The literate computing programming environment caused breakdowns.  |


**Malleability.** In the prototyping workshops, we found that there is still a gap between access to reprogrammable code and useful malleability—even for programmers. Even though participants in the workshops had access to the code of each of the tools that we prepared, it was still difficult for them to make meaningful changes to the code. While reprogrammability brings software technically in control of users it also can make users feel out of control by not knowing how to make use of it. Even though some participants were programming experts and we clustered the implementation of tools into packages, participants still struggled to understand the implementation of our tools. These challenges, however, were at least partially due to the limited time participants had to reprogram their applications—only around one hour. With this limitation in mind, using a reprogrammable prototype in prototyping workshops overall was promising as some groups successfully changed parts of the tools. Reprogramming tools, further, sparked discussions among participants and let them think about the tools in more detail.

Using the package management for the distribution of tools as packages, unsurprisingly, was a success. All groups managed to make use of the package management to extend the malleable word processor with the tools we prepared as packages. Compared to reprogramming it was more accessible for users, albeit, less powerful in the ways things can be changed.

A more general outcome of the overall co-design study was that participants had contrasting needs when it came to collaborative writing tools. These contrasting needs often need to be resolved outside of software, e.g., by having to adapt to the writing tool that is usable by all co-writers. Malleable software like computational media could open up ways for co-writers to tailor their tools to their needs, making it possible to collaborate on the same document, e.g., a paper or thesis, while using one’s preferred tools, e.g., a WYSIWYG (What You See Is What You Get) editor like Microsoft Word or a plain text editor like LaTeX.

**Shareability.** Shareability was an important aspect in this case study, because it allowed participants to work collaboratively together on their prototypes and also enabled us as facilitators to access the documents of group in order to support them or resolve bugs. Working together on the code, however, was not always beneficial: in some cases, participants disturbed each other when working in the same document together. For instance, when one participant was editing the text in the document and another tried to reprogram the code of a tool. The main issue of this was that the user interface of Codestrates v1 is partially synchronized. This means that which sections and paragraphs of a document are visible and which are collapsed is shared across all clients. Hence, when reprogramming different tools, participants
either were distracted by seeing the code their group member was editing or experienced “jumping code” due to the deletion or addition of code in other parts of the document.

**Distributability.** Distributability of documents was useful in this case study as it enabled participants to use their own laptops in the workshops. This not only reduced the equipment needed to facilitate such prototyping workshops but also enabled participants to use their preferred and personal working environment. Distribution of functionality or computation were not relevant for this case study.

**Computability.** Supporting computability, our prototype provided the computational environment together with the prototype. Participants did not need to install any development software on their devices but could directly get started with editing the prototype.

A big issue that we found in relation to computability, was that the programming model of Codestrates v1 was insufficient for collaborative and live reprogramming. Participants struggled to understand how the execution of code works and failed to match it with their familiar programming models, e.g., trying to run code “on load” of a webstrate. In Codestrates v1, however, code needed to be set to auto-run. Other groups had problems with applying changes: by re-running the code paragraphs they merely duplicated buttons for tools instead of updating the existing ones. This was due to the imperative nature of JavaScript and event listeners.

In addition, the literate computing environment of Codestrates v1 caused confusion among participants: The text document they were collaboratively working on was displayed in the same types of paragraphs and sections as the implementation code of these tools. Because of this some participants conflated the two and at times were unsure on whether they are writing text or reprogramming the tools. This slowed participants down during the reprogramming task and was in light of the already short amount of available time a problem.

### 6.3 CASE 3: GAME CHALLENGE

The third and last case I have worked on during my Ph.D. was centered around using computational media as a development platform in a game challenge. In the game challenge, participants were tasked to implement malleable multiplayer games using Codestrates v2. The findings of this section are based on Paper E (see Chapter 13).

#### 6.3.1 Case Background

This case was motivated by the desire to explore Codestrates v2 in a user study, where users use the platform for developing software. The
aim was to let users work open-endedly and evaluate the usability and understandability of Webstrates and Codestrates v2. By conducting a game challenge we aimed to make the study more playful and encourage participants to explore and be creative in their use of the Codestrates v2. Opposed to implementing an app, which users might connect with productivity and work, creating a game should be playful and without work or productivity tasks in mind. We, further, decided to let participants work in small groups in order to also gain insights on the collaborative capabilities and challenges of Codestrates v2 and whether some of the challenges of Codestrates v1 were overcome.

6.3.2 Case Overview

The game challenge took place entirely remotely due to the COVID-19 pandemic. The study consisted mainly of three activities: the coding time, a game demonstration meeting, and interviews. In the following paragraphs I will briefly summarize the conducted research. For a detailed description I refer the reader to Paper E, which is included as Chapter 13.

Participants. Participants were recruited through promoting the study in the Webstrates Slack workspace and on Twitter. We also directly invited potential participants that, for instance, worked with Webstrates or Codestrates v1 in the past. Participants were required to have experience web development and JavaScript. In total, 23 people participated in the study. 12 participants participated in the interviews and formed eight groups.

Procedure. The game challenge started with providing the participants with a design brief of the schedule and tasks of the game challenge. It included also introductory material about Webstrates and Codestrates v2. The game challenge consisted of four parts: making a small game, making the game multiplayer, making the rules or parts of the game editable through programming, and, lastly, making this editing process collaborative. There were no restrictions on what type of game participants could implement. Participants were also not required to fulfill all four parts, e.g., making just a multiplayer game was also sufficient, but we encouraged them to work on as many parts as possible.

During the three-week game challenge, participants worked individually or in small groups. Participants could decide for themselves when and for how long they would work on the game challenge. During the coding time, participants could ask for help — either from the facilitators or other participants — in a Slack workspace, which we set up for the game challenge. Initially the coding time was planned to last 16 days, later, however, we extended the time to 21 days as groups
required more time to work on their games. After the coding time, we facilitated a joint demo session where participants could present their games to us and the other participants. The demo session was held over Zoom and lasted one hour.

Lastly, we conducted semi-structured interviews with 12 participants. The interviews focused on participants’ experience in using Codestrates v2 as their development environment and inspiration for their game ideas. For participants who worked in groups we also focused on their experiences of collaborating with Codestrates v2. The interviews lasted between 15 and 75 minutes with most interviews lasting around 35 to 40 minutes.

**DATA COLLECTION AND ANALYSIS.** We recorded the audio of all interviews. For some interviews, we also recorded the video including screen recordings when participants shared their game during the interview over Zoom. We transcribed all interviews and translated those, which were conducted in Danish.

In the analysis, we used eight tensions, which we articulated after the game challenge, as guiding themes in a deductive coding analysis. The tensions came up as part of this paper when discussing how to analyze the data and when creating a chronology of the Webstrates and Codestrates development process since its beginning (see Paper E for more details). We analyzed the data in two stages: In the first stage, one author coded all interviews while three of the other authors coded each some of them so that all interviews were coded by at least two people. In the second stage, four of the authors discussed all interviews and their codes and went through them line by line. The discussions aimed to identify patterns and subthemes for each of the tensions that we used as the main themes.

**OTHER OUTCOMES.** As part of this project, we also revisited the history of Webstrates and the variety of efforts taken to realize the vision of computational media. We provided a summary of the Webstrates, Codestrates v1, and Codestrates v2 platforms and a chronology of their development over the years. Further, we present eight tensions that needed to be balanced during this development. The project concludes with six lessons learned from the endeavor to realize the vision of computational media. These contributions are included in Paper E in Chapter 13.

6.3.3 Perspectives on Computational Media

Using Codestrates v2 as a development environment in this case provided us with insights on the understanding of users of our realization of computational media. We reflected on the principles of computational media and found that realizing them can result in tensions
between an ideal solution and practical and technical limitations of creating an experimental prototype. Beyond this, we also learned about problems and potentials of malleability and shareability when it comes to collaborative programming. For instance, shareability and a not matching programming model caused breakdowns—similar to findings of my second case.

In the following paragraphs, I will summarize findings for each of the principles and facets of computational media. Table 6.3 provides a summary of these findings.

MALLEABILITY. During the game challenge, we learned about the importance of good versioning systems for malleable software: In Webstrates, each edit up to a keystroke level to code is versioned on the server. While this allows for finely-granular versions, it often does not encapsulate the semantics required to make the versions meaningful: As each letter written or deleted in the code of a document adds a new version potentially many versions are created. This issue was even amplified by creating new versions when the state of a game changed: For some groups, each change of a game’s state was persisted in the DOM and, thus, created a new version. The high frequency of some of these changes left some groups with five-digit version numbers, making rolling back to older versions practically unusable.

In making their games malleable, some participants thought about how much access they want to grant to players when they would edit rules of the game. For instance, some participants were worried that players could break the game when giving them access to re-programming. This highlights a challenge of malleability about how much access users should have to reprogramming so that they do not accidentally break their game or application.

SHAREABILITY. Similar to the previous case, shareability also was both bless and curse in this case: While it enabled participants to work in real-time together on their documents, which was especially useful for the remote setting of the study, it also caused breakdowns when working in the same document at the same time. For instance, when reloading a webstrate and running code that was not yet finished from being edited. This is also related to computability, which I discuss below.

When working on different features of their games in the same document, participants missed a way to keep their changes local, so that their teammates were not disturbed by running faulty or unfinished code. There is a need to provide mechanisms that support seamless switching between synchronous and asynchronous collaboration. Solutions could include, for instance, allowing code changes to be applied only locally and which could when finished be committed to the shared version.
<table>
<thead>
<tr>
<th>Principle</th>
<th>Potentials and Challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Malleability</strong></td>
<td>P: Versioning can help to improve feeling “safe” when changing software in order to rollback.</td>
</tr>
<tr>
<td></td>
<td>C: Versioning code changes and application changes in the same space can make versioning unusable.</td>
</tr>
<tr>
<td></td>
<td>C: It was difficult for participants to formulate how much access they want to give players in reprogramming.</td>
</tr>
<tr>
<td><strong>Shareability</strong></td>
<td>P: Allowed participants to work remotely together on the same document.</td>
</tr>
<tr>
<td></td>
<td>C: Issues related to running unfinished code.</td>
</tr>
<tr>
<td></td>
<td>C: No possibility to branch off and work on local changes.</td>
</tr>
<tr>
<td><strong>Distributability</strong></td>
<td>P: It made it easy for participants to get started programming without having to setup or install software.</td>
</tr>
<tr>
<td></td>
<td>C: Misunderstandings of participants on what and how synchronization with the server happens.</td>
</tr>
<tr>
<td><strong>Computability</strong></td>
<td>P: Bundling the computational environment and authoring environment with the document allows for on-the-fly changes of code.</td>
</tr>
<tr>
<td></td>
<td>C: The programming model forcing participants to refresh the whole codestrate after changes to run updated code.</td>
</tr>
<tr>
<td></td>
<td>C: Embedding a computational environment close to the application can cause context-switching breakdowns.</td>
</tr>
<tr>
<td></td>
<td>C: Some users prefer to use their own tools over the embedded development environment.</td>
</tr>
</tbody>
</table>

Table 6.3: Summary of the potentials and challenges, sorted for each of the four principles of computational media, found in the third case, the game challenge. Potentials are denoted with the letter P and challenges with the letter C.
Distributability. Distributability was beneficial as it allowed participants to open documents on their personal devices. This was essential, as the game challenge was conducted remotely. Still, distribution and persistence of documents on the Webstrates server caused some breakdowns: Participants had problems understanding the different ways in which state could synchronize the state of their games.

In one of the examples we provided participants with, we used a JSON fragment to synchronize the state as a model between clients. In another example, we used the signaling API. Yet another option, which especially participants that were familiar with Webstrates wanted to use, was to store state directly in the persisted DOM. Each of these methods is suited for different use cases: Storing state in a JSON fragment or the DOM was useful for persistent data that is updated infrequently, e.g., the todo items in a todo list. For transient data that updates frequently, like the position of a player, however, signaling was better suited as the other two methods would smother the Webstrates server with many edits that would need to persisted and that could cause conflicts. This confused some participants, as it was not clear which would be the “right” way to synchronize data.

The system should support the user in synchronizing state when developing in distributed documents, taking away the burden of the user. This issue is also something we worked on in the Varv project: Users do not need to think about how data is persisted, instead they only need to define what data should be persisted and Varv’s data stores take care of the how.

Computability. Bundling the authoring environment Cauldron with Codestrates v2 enabled participants to directly author their games within their codestrates. Participants mostly welcomed this feature, as it allows to perform small on-the-fly changes to their games without requiring them to bring their own IDE or code editor. Still, some participants would have preferred to use their own code editor and familiar tools to implement their game, because they missed common features like refactoring or autocompletion of code.

However, there were also issues with bundling the authoring environment Cauldron with Codestrates v2: As Cauldron is placed in the same space as the application in the web browser — by default it is docked to the right side (see Figure 5.2) — some participants had issues with context switching. For instance, when attempting to test their games, some participants accidently wrote in the Cauldron code editors as those were still in focus.

The most pressing issue in this case regarding computability, however, still was the unfitting JavaScript programming model used by Codestrates v2. Following an imperative programming paradigm, JavaScript is not designed to be modified and re-run at runtime. This can, for example, cause event handlers to be duplicated or old code to
be still running as it is not cleaned up — new code is merely added on top of the old code in the runtime instead of updating the old code accordingly.

But not only the executable code, also using HTML and the DOM for the user interface of an application or game suffered from this flawed model: Changing the HTML of a game in Codestrates v2 is merely changing the textual representation of the HTML, not its representation in the DOM. Thus, whenever the HTML is changed, the old representation in the DOM is discarded and the textual form is deserialized into the DOM again, which removes all event listeners or state in the DOM. Thus, once participants changed either the HTML or the JavaScript of their game, they practically always had to reload the whole webstrate before testing their game.

6.4 SUMMARY

In this chapter, I presented three case studies in which I used computational media prototypes and platforms. Through these cases, I could collect diverse insights related to the four principles of computational media and the use of computational media in general.

The first case looked into how a small group of researchers in the field of nanoscience used computation for their research: A case where researchers were using multiple devices in both the lab and office environment and were relying on computation for their work albeit not having the necessary training in programming skills. The computational labbook prototype was co-created with participants to create a possible future of software. The second case took a different angle on computational media as a tool for cooperative prototyping in co-design studies: A case of making use of malleability, co-located collaboration in using and editing a malleable word processor prototype. The third case yet again investigated another domain in which participants remotely collaborated on implementing a game from scratch with Codestrates v2: A case in which shareability and computability were essential, and which once more raised the issue of the inappropriate JavaScript programming model, which was not designed for collaborative and real-time changes. Especially the second and third case pointed towards common issues in the programming model used in Codestrates v1 and Codestrates v2, which I addressed with the Varv programming model.

Some of the insights presented in this chapter were relevant for the design of computational media, which was covered by the previous chapter. This demonstrates how the four principles of computational media can be used both as an analytical tool in these three case studies but, in turn, also as a generative tool when creating the platforms based on these insights.
REFLECTIONS AND DISCUSSION

In this chapter, I will reflect on the main results of my thesis, which were presented in the last three chapters. I will discuss the cyclic nature of my design work and empirical work and how they influenced each other. Further, I will relate computational media to common interactive objects and, finally, present limitations of my research that originate from the scope I chose for my Ph.D. project.

7.1 ITERATIVE CYCLES ON DESIGN AND EMPIRICAL WORK

As mentioned in beginning of Part I in Section 1.3, the results of the chapters on creating and using computational media, Chapter 5 and Chapter 6, were interrelated and evolved iteratively throughout my project. While some of the findings of the three case studies were addressed in the Codestrates v2 and Varv platforms, there are still open challenges that are unresolved. Figure 7.1 shows an overview of how the outcomes of the empirical case studies influenced the design work on the Codestrates v2 and Varv platforms.

7.1.1 Open Challenges

Besides the challenges that I worked on in the Codestrates v2 and Varv platforms, there were also other challenges that I did not work towards in these platforms. Most of these challenges are non-trivial and might require similar or more amounts of design and engineering work as Varv required. Hence, these challenges also open up directions for future work. Some of these challenges are not restricted to computational media but are also relevant for more conventional application-based software. The following paragraphs will provide an overview of four open challenges.

OFFLINE MODE. Adding an offline mode to Webstrates is a feature that was wished by the participants of the nanoscience lab study (Paper A). Being able to work offline and to keep data locally would be essential for them to meaningfully adopt such a system for their work. This aim is related to the facet decentralization of the distributability principle.

While this will be an important aspect in the future, Webstrates faces various technical challenges to in order to become decentralized. These challenges are, however, not only related to computational media but also to collaborative web apps in general. The CRDT (Conflict-
Figure 7.1: Overview of the connections between my empirical work and design work. Codestrates v2 occurs twice: first, as the outcome of the first two cases, and, second, as the platform used in the third case. Letters in the corners of cases and platforms specify the respective papers.
free Replicated Data Type) technology [160] is probably the most promising platform to build a decentralized version of Webstrates on top of. It has been explored in other domains like, for example, a collaborative rich-text editor [190] or a “collaborative corkboard” [127]. Adopting this technology in Webstrates, however, is a complex challenge due to the complex structure of the DOM and the high number of different operations that are possible in the DOM.

**Offloading Computation.** Another improvement that came out of the nanoscience study was the desire to offload computations from a document to another client or more powerful server application. While I implemented this feature for a specific use case in the computational labbook prototype, it was tailored towards the scripts of the nanoscientists and not a general purpose solution. Also the server application used in Videostrates [164], a prototype for collaborative video editing, was tailored specifically towards their use case.

Users should be in control of where computations happen. Ideally, this should work in an easy and fine-granular way, for instance, using a parameter on individual fragments in Codestrates v2 that defines where they should be executed. For example, locally in the current client, in a more powerful client, e.g., a desktop computer, or a server.

**Support for Asynchronous Work.** While Codestrates v2 and Varv enables synchronous collaboration on the code in fragments, where changes are immediately synchronized and visible to other collaborators, both platforms still lack capabilities to support asynchronous work in a better way. For instance, it would be desirable to be able to make local changes to code fragment which are not shared with others — both for JavaScript and Varv concept definitions. This would make it possible to test code locally without disturbing collaborators, that are currently in the same document.

Currently, a workaround to overcome this challenge is to create a copy of a document in Webstrates, perform the changes in the copy, and copy-and-paste the changes back into the shared document. This, however, is cumbersome and can cause conflicts if fragments were already changed by other collaborators. More advanced features such as being able to lock fragments to test them locally, might help to ease asynchronous work when collaborating.

**Versioning of Code and Application State.** Another domain that should be improved is the versioning of code and the application state. Versioning is important for malleable systems to allow easy backtracking of changes and to facilitate exploration without the fear of losing information. In Webstrates, versioning happens globally across an entire webstrate. Thus, changes to the code and changes to the application state are stored in the same place. Due to
the high granularity of versions in Webstrates, this often renders the versioning useless, because there is a high number of versions created. This was something that became evident in the third case study where participants had five-digit version numbers due to storing frequently changing state in the DOM.

Possible improvements to the versioning might include being able to view versions of the code and application state individually. A step more in this direction, might even allow to view the version history for each individual fragment in a document or, in Varv, being able to view the version history of individual concept definitions or instances.

7.2 Relationship to Common Interactive Objects

In Section 1.2, I summarized the CIO (Common Interactive Objects) project, which partially funded my Ph.D. project, and its main objectives. In brief, CIO investigates the use of common objects by people and communities in the multiplicity of their artifact ecologies. In this section, I will discuss the relationship between computational media and CIO in more detail. I will relate three principles and goals of the CIO project to computational media and describe similarities.

Empowering Users. The first objective of CIO I will discuss is the objective to “make common interactive objects that will empower users to better understand and develop the technologies they use” [66]. This objective relates to the vision of computational media to create a medium that is easy to use and understand — similar to how Kay [149] envisioned even children being able to manipulate the games they are playing and how diSessa and Abelson [78] envisioned “programming as a way for nonexperts to control a reconstructible medium.” But also in my vision of computational media I see and experienced understandability as a big hurdle for users to make use of malleability. For instance, in the second case study where even professional programmers had difficulties in understanding the code of the malleable word processor prototype.

While my work in this thesis, e.g., Varv, focuses on users with programming experience, I hope that by making software more malleable and easier to change, in the long-term, we can also empower non-programmers to better understand their software and modify it. I took a first step in that direction with the Varv project. By representing interactive behavior and computation in a declarative and structured representation, one of the motivations was to make software more understandable: Instead of crawling through imperative code, only a small set of primitives is used (concepts, schema, actions) and interactive behavior is defined in a more simple “when this happens then do that” format.
Still, Varv in its current implementation is only a first step and a proof-of-concept of declarative definition of interactive behavior. Improvements to the language design and, more importantly, the tooling are required to move closer towards the goal of enabling end-users to reprogram their software. Nonetheless, I believe that with these improvements Varv can drastically lower the barrier of modifying one’s software. While it is unclear whether this will be enough for users without programming experience to modify their software, I am optimistic it will support users with programming experience to better understand how a piece of software works.

**Malleability, control, and shareability over time.** The second objective of CIO I will discuss is to “carry out ground-breaking research regarding the technological basis of common interactive objects with focus on malleability, control and shareability over time” [66]. This objective focuses on the changes over time: Software and its use evolves over time and requirements and practices might change depending on the current situation. As Bødker et al. [46] describe it: “Common interactive objects have a history, and change over time. Change happens through use, and with interactive objects also through tailoring and re-programming.”

These changes over time can manifest themselves in *traces* [14, 27] of past routines, past activities, past objects, or other users. However, such traces might be missing, which can cause breakdowns. For instance, Avdic et al. [14] discuss how the voice user interface of smart speakers is often “traceless” and, hence, causes “breakdowns such as ‘Did the smart speaker not understand me?’” In malleable software, missing traces might also lead to similar breakdowns: When imagining a future in which software is malleable and can be changed by its users over time, then, in collaborative settings, one user might modify a piece of software to match the joint task in a better way. Without traces about this change of functionality, other users using this tool might encounter breakdowns, because the functionality changed without their knowledge.

In systems like Varv, which support accretive extension by changing interactive behavior through the addition of “layers” of code and without changing the original code, such traces might also be a challenge. While the layers of changes can act as traces into how software changed over time, it can be challenging to understand the bigger picture of how a piece of software works if only traces are visible.

**Common objects and artifact ecologies.** Lastly, I will discuss the principle “multiplicity of artifact ecologies” [27] of CIO. This principle describes that common objects are used in a variety of “activities, configurations of people, applications, and device” [27]. Hence, common objects should embrace these different constellations and sup-
port transitions through them. Software created with computational media, which is shareable and distributable, can support this principle: Distributability makes it possible to use the same document across different devices and shareability provides means for shareable documents and collaboration, allowing different groups of people to work with the plan—both in synchronous and asynchronous scenarios.

7.3 LIMITATIONS

In my Ph.D. project, I investigated computational media as a vision for software, which is in stark contrast with the current application-centric model. Investigating how such a vision of software would operate in the real world, e.g., in a company or community, would require a foundation of production grade software created with computational media, which would need to be deployed with users over an extended period of time. Such an endeavor would be not feasible within the scope of a Ph.D. project, due to missing development resources to create robust, production grade software, missing time resources, and missing access to participants that could afford to use such software over an extended period of time for their actual work.

Thus, in my Ph.D., I focused on a slice of this vision: What principles computational media should have on a technological level, how can these be achieved in practical implementations, and what potentials and challenges do they come with for individual users or small groups of users. I did this by, first, creating experimental platforms for building software, i.e., Codestrates v2 and Varv, and prototypes that are build for specific cases, i.e., the computational labbook and the malleable word processor. Second, I used these platforms and prototypes in small-scale empirical case studies. This allowed me to investigate aspects of computational media in a scope which was feasible in my Ph.D. project. However, this also meant that I had to remove some aspects of the implications of software as computational media from the scope of my project.

First, my prototypes focused on how to create software that follows principles of computational media while keeping the emphasis on new ways of interacting with software and modifying software. However, my work has shown that security, privacy, and data provenance are important aspects as well. For instance, for the nanoscientists in my first case study, who were concerned about data provenance in malleable documents, or for participants of the prototyping workshops in my second case study, who were skeptical about how malleable and extensible software would be maintained over time by the people who create the software and its extensions. I acknowledge these concerns and that more future work in these areas is required to explore how these challenges would affect software created in computational
media over time, e.g., when software gets updated and extensions become incompatible.

On the user side, my empirical studies focused on individual users and small groups of users. However, as the work on the Buttons system by MacLean et al. [197] demonstrated, the culture around software is just as important for successful modification of software as the technical mechanisms in place. Hence, to fully investigate how computational media would affect software and how users modify it, one would need to create a culture in which this is the norm. This, again, would require larger-scale studies with bigger groups of users over time. diSessa [77] described this as the “cultural level” of exploring computational media, after a “technological”, which focuses on creating platforms and artifacts, and a “cognitive level,” which focuses on small-scale experiments.

From a political or economical standpoint, software based on computational media could have interesting implications. For one, if the source code of software is shipping with software and is modifiable by the user, questions on how to prevent pirating of software or how to handle licenses of modifications are inevitably raised. Nouwens [232] describes that with “the business model change from software as a ‘free’ package to software as a tradeable product, there was a shift in responsibility over the quality of that software, and a subsequent appropriation of control over the software by its producers in order to better manage those responsibilities and ensure continued profit.” Hence, it might not be in the interest of companies to increase users’ control over their software. This, however, is a challenge in itself and beyond the scope of my Ph.D. project.
In my thesis, I revisited the vision of computational media, which presents an alternative foundation of software to the predominant application-centric model. I explored the vision in three domains: theory, design, and empirical work. I started by offering my definition of computational media, its four key principles malleability, shareability, distributability, and computability, and a summary on how it compares to similar existing visions. Next, I presented how these principles can be used as a generative and analytical lens in creating computational media platforms. I did this by describing the existing platforms Webstrates and Codestrates v1, on which my research builds upon, and by contributing two platforms of my own: Codestrates v2 and Varv. Finally, I demonstrated how these platforms can be used in three empirical case studies over the course of my project: the nanoscience lab study and the collaborative writing study, which both used Codestrates v1, and the game challenge, which used the Codestrates v2 platform.

In the remainder of this chapter, I will summarize my contributions, relate them to my research questions, and provide an overview of possible directions for future work.

8.1 Thesis Contributions

In my Ph.D. project, I set out with the goal to explore computational media as an alternative to the application-centric model. This exploration is important in order to better understand the possibilities of these alternatives and how they might complement application-based software. I explored computational media guided by three research questions that aimed to investigate computational media in three strands: theory, design, and empirical work. Chapter 4 (Defining Computational Media), Chapter 5 (Creating Computational Media), and Chapter 6 (Using Computational Media) each contributed to one of these three strands. In the following paragraphs I will summarize my contributions to each of these strands.

Theory Contributions. Drawing from existing visions of computational media and results from the empirical cases I conducted, I synthesized a list of four key principles and related facets that computational media should embody: Malleability describes how documents should be modifiable by the user, e.g., through extension, configuration, or reprogramming. Shareability focuses on addressable...
documents, that can be shared and allow for real-time collaboration while making it possible to also support asynchronous work. Distributability enables documents to escape the scope of applications or devices, instead they should be able to be used across different device types and operating systems, and also make it possible to distribute functionality or computation across devices. Computability, lastly, describes the possibility to make use of computation in documents, so that custom computation can be used inside a document, the user interface can be addressed, and the programming model fits the medium.

These principles and facets describe mostly technological qualities or capabilities of computational media and, hence, can be used in a generative way when creating computational media platforms or prototypes. Additionally, they can be used as an analytical and critical lens when analyzing software or the use of software. For instance, in the nanoscience lab study (Paper A), I used them both to analyze qualities of the computational environment of the researchers and to design the computational labbook prototype.

Design contributions. The first part of my design contributions are two experimental prototypes that I used in two of my case studies. Building on the platforms Webstrates and Codestrates v1, I created two prototypes to explore concrete realizations of computational media. In the nanoscience study, I developed the computational labbook prototype, which enabled scientists to envision software which combines the flexibility of command line scripts with the usability of user interface applications. In the collaborative writing study, next, I developed the malleable word processor, which enabled to add, remove, and modify additional features as packages. It, further, enabled reprogramming of these tools right within the document.

The second and main part of my design contributions are the design and development of two platforms for the creation of computational media documents. Using the insights from the empirical case studies, I developed two platforms, that advanced the capabilities to develop applications with computational media. Informed by the first two case studies and prior work, the first platform I created is Codestrates v2, which focuses on generalizing ideas from Codestrates v1 and provides a development platform for Webstrates, including a package manager, an execution engine, and the authoring environment Cauldron. The second platform I created is Varv. Varv is the main design contribution of my Ph.D. project and aims to overcome the lack of a matching programming model for computational media or — more general — for reprogrammable and collaborative software. Its decoupled architecture and declarative representation of interactive behavior in concept definitions, makes it possible to create a live programming environment that shares state across clients. It, further,
allows for accretive extension of interactive behavior by its capability to merge multiple concept definitions.

**Empirical Contributions.** Throughout my project I conducted three empirical case studies using computational media prototypes and platforms, and contribute findings of these studies, that advance the understanding of computational media and its use. The *nanoscience lab study* demonstrated potentials of computational media as a suitable platform for domains that require high flexibility in their computation but where computation is used by users that are not skilled in programming. The *collaborative writing study*, next, explored more actively how users can be directly involved into the activity of reprogramming existing software in a workshop setting, highlighting the need for a more fitting programming model. The last study, the *game challenge*, expanded on this and investigated how the Codestrates v2 platform can be used by small groups of people to develop reprogrammable multiplayer games.

The second case study, further, highlighted challenges of using the literate computing paradigm and the computational-notebook-inspired user interface in Codestrates v1. These findings were leveraged to design Codestrates v2 as an iteration of Codestrates v1. Next, especially the second and third case studies demonstrated the shortcomings of using JavaScript as the programming model for our computational media platforms Webstrates, Codestrates v1, and Codestrates v2. This, in turn, motivated the work on Varv to overcome some of these challenges in a programming model that was design from the ground up for inherently extensible software.

**8.2 Directions for Future Work**

As mentioned in the introduction, I see my research in this thesis and computational media not as a definitive solution that should replace applications altogether. Instead, I see it as a stepping stone towards an alternative to application-based software. Hence, based on the work presented in this thesis, it is possible to expand into various directions in future work. In Section 7.1, I already touched upon some of the open technological challenges, which the empirical studies brought up. But also especially the Varv platform opens up a variety of new possibilities to investigate reprogrammable software or collaborative programming practices. The following paragraphs will summarize some directions this future work could take.

**Investigating the Usability of Varv.** While my co-authors and I demonstrated the expressiveness of Varv through multiple example applications and two case studies in Paper F, we did not yet involve users into the evaluation. To better understand what repre-
senting interactive software as a declarative data structure implies, a next step needs to be to evaluate the platform with users. Here, multiple directions can be taken: To evaluate the usability of the concept language and the Webstrates-based implementation of Varv, a controlled user study with participants with programming language could inform whether or how Varv might improve the efficiency of reprogramming applications and whether it lowers the threshold to do so. A low enough threshold to engage in reprogramming could also allow users with little or no programming knowledge to meaningfully modify their applications.

To evaluate how software that is inherently malleable might change the way people think about modifying their software, another more longitudinal study in the wild might be conducted. For this, applications that participants use on a everyday basis could be re-implemented with Varv and given to participants for several weeks. This could investigate whether the possibility and the lower threshold to perform changes motivate users to modify their software more often. This would, however, require to create a meaningful and reasonably large application with Varv.

Lastly, to evaluate the collaborative aspect of Varv, a controlled study with multiple users could investigate, whether the live-updating nature and shared state of Varv allow multiple participants to modify an application collaboratively in real-time. It would also be interesting to investigate whether the shortcomings of divergent state from Codestrates v2 can be overcome with Varv. This could provide interesting insights for possible future work regarding asynchronous work (see below).

**Asynchronous Work**

ASYNCHRONOUS WORK AND VERSIONING. Two of the open issues that my current platforms do not yet solve are better support for asynchronous work and versioning (see Section 7.1). In the game challenge study, participants, who worked together in groups on a game, struggled when other group members worked on different features of the code at the same time: Changes could break the code temporarily and with that prevent other users from testing their respective changes. Computational media requires mechanisms to support asynchronous work when collaborating in these settings, for instance, a solution might be to let users “lock” code fragments and let them perform local changes that are only applied to the current client. Once a feature is finished, the fragment could then be “committed” and unlocked again, so changes are visible and activated for other users.

Similarly, the versioning of Webstrates was opaque for users in the game challenge, making it difficult to roll back changes in a meaningful way. The conflation of application versions and code versions in the same space increased this issue even more. Future work could investigate better ways to version code and application state,
and provide users with ways to filter them for either the application versions or the code versions. Taking this even a step further, code could be versioned on a fragment level, so that versions of a fragment can be rolled back—similar to code files in a Git-versioned project. With Varv, similar versioning could be imagined for the state of concept instances in data stores where, for example, the versions of individual concept instances could be explored and rolled back.

**MECHANISMS FOR INTEROPERABILITY** Varv defines interactive behavior and state in a structured format. Concepts have a schema with properties and instances of these concept add values to the properties. While it is possible to use a copy of an application and changing the behavior for the given concepts and their respective schema, combining actions from concepts with different schema is non-trivial. For instance, consider a todo list app, which stores todos with a title string property and a completed Boolean property. Another todo list app or Kanban board app might store tasks with a name string property and a state string enum property with the states backlog, in progress, and done. Actions that toggle the completed property cannot easily be mapped to the state property and vice versa. The work on Cambria by Litt et al. [186] deals with such problems by adding *edit lenses*, which make it possible to map from one schema to another in a bidirectional manner. It would be interesting to combine the programming model of Varv with such edit lenses to create a platform for even more interoperable software.

### 8.3 CONCLUDING REMARKS

*Reducing effort is incredibly exciting because it doesn’t just save time, it changes who engages in a task, and when! When we perceive a high amount of effort relative to the return, we prejudge a task as not worth the effort. Conversely, when a task is perceived as easy, we perform the task in more contexts.*

— Mike Bostock [56]

I hope that my thesis about the vision of computational media and steps towards realizing such a vision prompts readers to challenge the taken-for-grantedness of application-based software. During my project, I enjoyed exploring this vision and asking “what if” questions that fundamentally challenge how we think about software and the act of modifying it—especially in Varv. While I do not expect — and do not think it should — that computational media could replace application-based software altogether, I see it as a promising software platform to live besides application-based software: In particular for fields like the nanoscience group of my first case study, which does not yet have established application software for their tasks, computational
media has potential to support their work and make computation more accessible. For this to happen, however, we also need users and cultures in which modifying one’s software is more established.

Similar to the quote from Mike Bostock from above in which he describes how he thinks about lowering efforts to create visualizations, I hope that lowering efforts to modify one’s software and use computation in projects like Varv also changes who modifies one’s software and how often they do so. Over time this might be able to create a culture in which using custom computation and modifying one’s software is the norm. Finally, I also hope that this work inspires others to conduct more research in this field and to explore other ways in which a vision like computational media could be realized.
Part II

PUBLICATIONS AND MANUSCRIPTS
BETWEEN SCRIPTS AND APPLICATIONS: COMPUTATIONAL MEDIA FOR THE FRONTIER OF NANO-SCIENCE*

—

MIDAS NOUWENS, Aarhus University, Denmark
MARCEL BOROWSKI, Aarhus University, Denmark
BJARKE FOG, Aarhus University, Denmark
CLEMENS N. KLOKMOSE, Aarhus University, Denmark

ABSTRACT

The popularity of computational notebooks heralds a return of software as computational media rather than turn-key applications. We believe this software model has potential beyond supporting just the computationally literate. We studied a biomolecular nano-design lab that works on a current frontier of science — RNA origami — whose researchers depend on computational tools to do their work, yet are not trained as programmers. Using a participatory design process, we developed a computational labbook to concretize what computational media could look like, using the principles of computability, malleability, shareability, and distributability suggested by previous work. We used this prototype to co-reflect with the nanoscientists about how it could transform their practice. We report on the computational culture specific to this research area; the scientists’ struggles managing their computational environments; and their subsequent disempowerment yet dependence. Lastly, we discuss the generative potential and limitations of the four design principles for the future of computational media.

9.1 INTRODUCTION

We are in a paradigm of software as applications — graphical interfaces to transform data using pre-defined tools — rather than computational media — software which allows you to create, execute, and edit computations that can operate on anything, including the software itself. Tantalizingly, however, there are recent commercial products and research

prototypes that run counter to this application paradigm. The computational notebooks that have become popular in data science, digital journalism, and education, for example (e.g., Jupyter Notebook [256] or Observable [236]), allow users to mix traditional word-processing functionality with executable code in the same perceptual space.

As seeds for a paradigm shift of software, however, computational notebooks are programming environments with some application-like qualities, rather than that they are GUI applications that also allow for writing and executing computations. As such, they treat code as the primary object of interest and expect users with software engineering skills (e.g., version control, dependency management). We want to expand this reimagination of software as computational media and explore designs that support computer users who are dependent on computation for their activities, but who are not trained as software engineers.

We collaborated with a group of biomolecular nanoscientists whose scientific practice and progress rely on using in-house developed computational tools, but who do not have a formal programming education. We engaged them in the participatory design of a computational labbook that helps them carry out a key part of their work, namely the computational design of new RNA structures. The prototype and participatory design process were guided by the concepts computability, malleability, shareability, and distributability; four proposed principles for computational media derived from Klokmose et al. [162] and Rädle et al. [257].

We address the following research questions:

1. How do biomolecular nanoscientists use computation in their work?
2. How can computational media, rather than applications and scripts, better support their work?
3. What can we learn about computational media at large by designing for biomolecular nanoscientists?

Accordingly, our main contributions are: (i) an empirical account of the computational activities and challenges of biomolecular nanoscientists; (ii) a high-fidelity computational labbook prototype for the domain of biomolecular nanoscience; and (iii) a deepening of the four principles for computational media rooted in real-world work praxis.

9.2 RELATED WORK

We examine the potential and principles of computational media in a scientific practice. Related work includes computational tools in the sciences and work on computational media.
9.2 RELATED WORK

9.2.1 Lab Notebooks and e-Science Tools

In the sciences, the laboratory notebook is often seen as the cornerstone of research, and efforts to create electronic lab notebooks (ELNs) are plenty. Commercial ELNs are largely wiki-based and oriented towards particular considerations such as transparency, data management, and ease-of-use (e.g., Confluence [13], BIOVIA [15], and Labguru [173]). Some academic work has explored other aspects of electronic laboratory work. Inspired by the Semantic Web, Talbott et al. [284] (among others [319, 320]) have focused on data provenance as a key factor in lab work. Others have aimed to bridge the gap between physical and digital materials by providing an interactive tabletop in the lab space [283], augmenting the work-space with mixed-reality “interactive paper” [203], or using digital photographs to couple notebooks and external materials [314]. Oleksik et al. [238] have investigated the artifact ecologies of laboratories and how scientists appropriate existing off-the-shelf note-taking software and the inherent clash between stability and flexibility, providing guidelines to how ELNs might be designed [239].

Most of the work on ELNs, however, has sought to more or less replace physical notebooks with digitally enhanced replicas. These enhancements are often web-based and collaborative, offering tools for cross-referencing, tagging, and visualization [95, 265, 277, 297]. While these are admirable efforts in their own right, we believe that looking more closely at the potential of computational media can fruitful directions for re-thinking laboratory notebooks to be the locus of scientific computational activities — as also proposed in [165].

The natural sciences have been the targets of efforts to augment the scientific process with computational capabilities, in particular when their objects are large-scale and complex data, e.g., genomic data, which often is referred to as e-Science. Smith [279] argues that contemporary scientific work is increasingly reliant on full-fledged software engineering abilities. Surveying the literature on e-science reveals a large body of work that focuses on the technical infrastructure for distributed and grid computing. However, there is also work that addresses challenges similar to some of those we observe in our studies: Holdgraf et al. [133] explore how to facilitate sharing of computational environments, Östberg et al. [244] study how to ease distribution of computations by raising the required level of abstraction, and Harrison et al. [128] developed a standard for sharing scientific computational workflows. Yet, their aim is not to challenge the dominant software paradigm but to create practical computational tools to support scientific work. We argue that many of these identified challenges are systemically tied together and can collectively be addressed by a change in focus from scripts and applications to computational media.
The idea of software as a dynamic or computational medium was conceived by Alan Kay in his famous Dynabook vision [149]. diSessa exemplified a computational medium [77] with his Boxer programming environment [78] that allowed users to build software and do computation in the same medium as they write text. diSessa contrasted the “monolithic, nonmodifiable applications” with computational media, perceiving the latter as tools that can be organically enriched and extended [77] (see Figure 9.1).

In recent years, the proliferation of computational notebooks, e.g., Jupyter Notebook [256], is emblematic of the perceived and actual benefits of computational media. In 2017, over a million Jupyter notebooks had been shared on GitHub [267]. Computational notebooks has has also recently become an academic object of study in HCI (e.g., [129, 153, 154, 266, 267]). Computational notebooks are not ELNs by another name, rather they are computational environments that enables weaving “a narrative directly into a live computation, interleaving text with code and results to construct a complete piece that relies equally on the textual explanations and the computational components” [249]. They are particularly designed to support replication in computational sciences [166]. Computational media do not necessitate that the user becomes a programmer. A particular group of these media can be termed no-code computational media. No-code computational media embody computational capabilities but do not require that the users themselves write program code. One such example is Notion [231], a knowledge repository environment that enables users to utilize computation without writing code. Another example is Coda [64].

Webstrates [162] is a platform for developing dynamic or computational media inspired by Kay’s early visions of personal dynamic media. Webstrates is based on a simple change to the mechanics of the web, where a web-page is made collaboratively editable directly from the browser. In Webstrates there is no technical distinction be-
between editing content and the scripts defining the functionality of a page. This means that the traditionally hard distinction between development and use of software is blurred. Codestrates [257] is an authoring environment for Webstrates and builds on the computational notebook metaphor. However, it allows for the development of application-like collaborative software, and is reprogrammable and extensible from within. Webstrates and Codestrates are document-centric in the spirit of Boxer.

Webstrates is based on the principles of shareability, distributability, and malleability. Shareability refers to the users’ ability to share and collaborate seamlessly on multiple types of data within a document, synchronously and asynchronously, with themselves and multiple people, using their own personalized views and tools. Distributability points to the ability to easily distribute documents, interactive elements, and computation across multiple and heterogeneous devices and platforms. Malleability signifies the system’s ability to be continuously changed and appropriated by users in personal and idiosyncratic ways. We used these principles as guiding design principles for our participatory design process with the inclusion of the principle of computability, which is left implicit in previous work on computational media. With computability, we refer to the ability to develop, edit, and execute simple and complex computations on data that exist in the same perceptual space as the code. We used Webstrates and Codestrates to create our prototype.

9.3 Method

Our aim was to explore computational media as an alternative vision of software to traditional applications and scripts. To make this actionable, we looked for use cases where computation was an integral part of users’ activities, but where users were not employed or trained as a programmer; activities where using and developing a piece of software blur, and users who work close to the code but did not necessarily author it.

We collaborated with a group of biomolecular nanoscientists because they (a) do complex knowledge work mediated by digital tools and materials, some of which are developed by lab themselves; (b) are computationally dependent but not trained to be computationally literate; (c) have a history of experimenting with new research tools and processes; and (d) were willing to participate and easy to access.

Our research consisted of three core activities over two years: (1) observations and interviews; (2) participatory design of a prototype; and (3) in-situ interviews while using the prototype (see Figure 9.2 right).
9.3.1 Participants

We collaborated with the Andersen Lab for Biomolecular Design\(^1\) which is part of the Interdisciplinary Nanoscience Center\(^2\) at Aarhus University. The nanoscience center is a consortium of fifteen departments and faculties across two universities, housing twenty-five research groups. At the time of the study, the lab consisted of one principle investigator (PI), eight postdoctoral researchers (one remote), and four Ph.D. students. All members of the lab except one were

---

1 Andersen Lab: https://bion.au.dk (Retrieved February 22, 2022)
2 iNANO: https://inano.au.dk (Retrieved February 22, 2022)
involved in the study to different extents. The process overview in Figure 9.2 to the left shows the participant engagement in the different activities in the project.

9.3.2 Observations and Interviews

To gain a phenomenological understanding of the work practice of the participants, their use of computational tools, and the socio-technical context of the lab, we took a contextual inquiry-based approach, which totaled five whole days of observations at the lab and six formal in-context interviews. Before starting our observations, we were introduced to the research group at their weekly meeting. We explained our overall goal and asked for consent to record their work. The participants were introduced to our general research agenda and we demoed previous prototypes of computational media that we had developed. We followed two participants for the entire duration of a particular experimental phase—each lasting between a day and a half— and opportunistically observed and interviewed the other members of the lab when appropriate. During the experiments we tried to get a general understanding of the participants’ work practice and asked them to walk us through the different steps of their experimental process, why they were doing it in the way they were, what problems they run into while doing their work, and what they would like to see changed. In particular, we asked them to report on the variety of computational tools they used and how they worked together. During the formal interviews, we used the critical incident technique [104] to ask the participants about recent or memorable technology breakdowns. The data collected in this phase included field notes, photos, videos of participants interacting with equipment and software, and audio recordings of interviews and workshops. The interviews were transcribed pure verbatim.

9.3.3 Participatory Design of a Possible Future Prototype

To help the participants reflect on how computational media might or might not be useful for their work practice, we wanted to establish a possible future [269], that is, use prototype development to co-create a future vision of a computational laboratory notebook “empirically researchable in the present world” [269]. The aim of this participatory design process was to encourage the participants to iteratively reflect on their current computational practice, crystallize ideas into prototypes, and give them hands-on experience with a possible future lab notebook.

The participatory design process started with a full-day workshop that focused on collectively imagining the future of digitally supported biomolecular nanoscience. We presented the empirical data
we had collected and asked the participants to confirm and expand on our observations. Grounded in these accounts, the participants wrote detailed scenarios of one particular, common activity of their work practice. They were asked to bring up the different digital and analog materials they used, the breakdowns that occurred, and the workarounds they employed to deal with them. We picked two of these scenarios and ideated in two groups on how the scenario could be better supported digitally in the future. We chose one of these two ideas to design and develop into a prototype. The idea was chosen based on its level of significance to their research, and its technical feasibility to implement.

We focused on the first part of their experimental workflow, which is the design phase of RNA structures. We conducted four meetings with researchers of the laboratory and a collaborator in the US to iteratively design and develop the prototype.

Using previously conducted fieldwork as a basis, we implemented the initial iteration of our prototype. It featured rudimentary functions like the displaying of a 3D representation of an RNA molecule or a drawing canvas for sketches. We used this first iteration as a way to communicate our ideas during the first meeting with two researchers of the lab. This meeting was used to get an initial reaction towards our idea of computational media and to clarify the exact steps of the workflow of one of the researchers.

After developing the structure of our prototype, during a second meeting, one of the researchers showcased how the tools and scripts are used in practice. This showcase was followed by a discussion on how and with what features the researcher could imagine this workflow using a computational medium. The third iteration of the prototype, which implemented various scripts and tools the researchers use, was discussed in a meeting with three researchers of the lab, two of whom were introduced to the prototype for the first time during that meeting. After a brief introduction to the prototype, all three researchers used it with one of the molecule designs they were working with at the time. Based on the feedback and ideas of the three participating researchers, the prototype was iterated another time. The fourth prototype was used as the basis for the in-situ interviews described below.

9.3.4 In-situ Interviews While Using the Prototype

We conducted in-situ semi-structured interviews with three participants while they used the prototype to design a new RNA structure. We chose these three participants because they, at the time of the study, had concrete work they needed to do which the prototype was specifically designed to support. The three independent sessions with the participants started with us giving them an introduction to the goals
of the project and the process so far, a brief mention of the history of these types of software tools, and a few words on what HCI research is. We introduced the participants to the four principles that the prototype was built upon: shareability, malleability, distributability, and computability. We did this to allow them to reflect on these, and for them to have a vocabulary to talk about the prototype with. After the introduction, which lasted between 15 and 20 minutes, they received an in-depth demonstration of how the prototype worked.

We began the interviews by asking the participants to design an RNA structure using the prototype. The prototype was used as a springboard into a discussion of their computational work, their thoughts on the four principles, and a general reflection on computational media in relation to their work. The interviews dealt with their general software problems, further explaining and discussions of the inner workings of the prototype, and even co-debugging their pre-existing scripts that had been imported into the notebook prototype. The interviews were guided by a number of questions we had prepared, e.g., “When was the last time you tweaked a script?” or “Do you have any shared documents or resources in the lab?” The interviews lasted 82 minutes, 57 minutes, and 90 minutes, respectively. All authors coded them independently (open coding), focusing on statements related to the four principles of computational media. We collected the codes and examples on a whiteboard and discussed how to arrange them in themes over three meetings.

9.4 Findings

The findings are structured as follows: first, we give an overview of the nanoscience laboratory and explain the work and environment of the participants. Second, we describe three defining computational characteristics of biomolecular nanoscience that affect the participants. Lastly, we present the design of our computational labbook prototype and how it relates to the participants’ work practice and computational challenges.

The findings are presented in an anachronistic order. The computational characteristics we describe came out of co-reflecting with the participants while using the prototype for a realistic work task. However, we present these insights before we describe the prototype’s design, because we believe this makes it easier for the reader to understand and appreciate what challenges the prototype tries to address.

9.4.1 Overview of the Lab

Nanoscience is the study and application of materials and structures on the nanoscale, a transdisciplinary research category that can be found in fields such as chemistry, biology, physics, engineering, and
material science. The research group under discussion here is one of the world’s leading molecular biology labs specializing in the origami method: a way of creating molecular structures using DNA, RNA and proteins as building blocks which can assemble themselves into non-arbitrary shapes. This allows researchers to create “nanorobots”; structures that can perform pre-programmed tasks such as turn into a smiley, act as biosensors, or (hopefully in the future) deliver drugs to specific parts of the body.

The experimental process of this area of research can be roughly divided into two parts: the “dry” part, which includes the open-ended and creative design of new structures; and the “wet” part, which includes creating those structures in the lab and performing experiments on it to see how it behaves.

For the wet work, each of the researchers in this group has their own assigned workbench in the laboratory (see Figure 9.3). The lab is crowded with scientific instruments they use for their experiments (e.g., vortex centrifuges, electroporators, non-confocal laser scanners). Because they work with genetic material, they have to be careful not to accidentally contaminate their samples. They wear latex gloves in the lab, and there are strict rules about “glove-on” and “glove-off” equipment. They also limit the number of things entering or leaving the lab as much as possible to protect the environment, and anything brought from the outside is only allowed to be placed in specific taped off sections on the lab bench.

The dry work is predominately done at a desk in the office. Over the years, each researcher has accrued an ecosystem of computational
tools they feel most comfortable with. Some of them have invested significant time designing their ideal software environment, with scripts to automate parts of their tasks, sophisticated Excel spreadsheets, and carefully chosen commercial and open-source web and desktop applications. Some of the researchers are provided with a laptop by the department, others simply bring their own device. Because of the restriction on moving things into and out of the lab, to use a computer in the lab means sacrificing that computer to that space, resulting in some of the researchers having one high-quality device for their office work and a second “disposable” laptop for the laboratory. All team members are also supplied with Apple iPads as part of the PI’s efforts to digitize the lab and experiment with new ways of working.

There are two computational tools that are absolutely fundamental for the lab, the knowledge it creates, and the way they organize their work. The first is their ELN: a wiki-based collaborative software called Confluence. The notebook is the linchpin of the natural sciences, used to plan experiments, record research results, and document analyses. It plays an important part in ensuring scientific integrity and settling intellectual property disputes. The Confluence ELN — a digital remediation of the traditional paper notebook — started as a grassroots project in this group, but was later picked up by the university administration and is now used by most of the labs. Each member of the group has their own webpage on the ELN, which they use to plan and report their experiments, present results during weekly meetings, hand over projects between researchers, and onboard new lab members. The second crucial computational tool is their repository of in-house developed Perl and Python scripts, which the researchers use to compute RNA structures, generate 3D visuals, and reformat outputs to be more readable.

These scripts were written by one particular postdoc who taught himself how to program, and they are used in good faith by other lab members who have fewer or no programming background. Without these scripts, the lab would not be able to do RNA origami research, since this is a frontier of science that is still being invented.

9.4.2 Computational Characteristics

We identified three characteristics of computational biomolecular nanoscience that shape how participants structure their work, their psycho-social working conditions, and scientific knowledge they produce related to their computational tools.

9.4.2.1 Computational Culture

Biomolecular nanoscientists have a distinct relationship with computers and computation compared to other, related scientific fields. Reflecting on the tools they use, P3 remarked:
Our lab is derived from molecular biology, so our methods are also from that field. We use “black box” machines a lot, where you put in your sample and just press start and you get your data. Physicists [on the other hand] like to have their own equipment they can tinker with. (P3)

The sophistication of those “black box” machines — whether hardware or software — differs quite significantly depending on the particular research questions the scientist focuses on. If the area of study is well established, so is the computational support. But going beyond the frontier of an established type of research means stepping into a tabula rasa of software. P10, for example, was confronted with this jarring difference when switching from analyzing RNA structures in their previous job to designing them when they joined the current lab:

We didn’t execute a whole bunch of codes in our previous lab. I was more using pretty well established software with a nice-ish GUI [...] I was still doing RNA work, but I wasn’t doing any design of RNA sequence or anything like that. So depending on the research you’ll have either really well supported workflows ... or nothing ... or homebrew scripts. (P10)

Switching research focus, then, also means a shift in the kind of computational competencies required of the researcher. For the ground-breaking RNA origami that this lab does, there are no well-established tools available. The lab members have to cobble together scripts in different languages, some of which are built in-house and others they found online. When asked if they could do this kind of research without those scripts, P11 answered:

No, no I couldn’t do it. [...] I could do other kinds of science, but this is what is giving us like, a lot of chances to create new structures. [...] I see sometimes other people designing RNAs manually in some papers ... that’s crazy. Sequence design manually? Do you know how crazy that is? (P11)

The friction comes from the fact that the software skills necessary to understand, run, and debug those scripts are not (yet) part of the formal education leading up to this kind of research, nor is it considered part of the professional culture to pick them up. Biomolecular nanoscientists with the computational literacy to produce or modify scripts are “pretty rare” (P10) and learning how to program is considered “a whole other career” (P11) rather than an integral part of their job. As a result, the development and maintenance of the computational tools vital to the participants’ research rests on the very precarious foundation of a once-in-a-blue-moon computationally literate graduate student, postdoc, or research assistant.
In summary, the computational culture of biomolecular nano-science is one where there is a dependency on computational tools to do the work, but not a tradition of training scientists in the development, deployment, and maintenance of them. This is fine when doing established research using GUI software, but becomes very restricting when working on a frontier of science where the tools have to be invented. There is an incongruous need for the flexibility of scripts that allows researchers to invent new methods, but the usability and convenience of applications. A computational medium would have to address this gap between flexibility and usability.

9.4.2.2 Computational Environment

The biggest challenge for the nanoscientists was not a lack of coding skills, but having to manage the computational environments in which to execute code. Because there are few computational tools that support their work, the participants use any software—whether an open source system published on GitHub or a script sent by a colleague from a different lab—regardless of language, dependencies, or quality of documentation. Installing, maintaining, and debugging the various environments that these tools rely on, however, has proven to be categorically prohibitive. Issues include terminal commands that change across operating systems, having to set up virtual environments to run scripts, fixing corrupted installations of language interpreters, etc.

The consequences of these challenges are that they work with partial outputs, accept bottlenecks, and even abandon certain research approaches. For example, P8 explains how they had to give up on a popular tool because they could not get it to work in Windows.

> I was trying to get this software that everyone was using [...] and I couldn’t get it to work [...] Probably because it’s for like Linux and my Windows just didn’t [...] so I had to either do a virtual machine, get Linux, and do it from there [...] but then I just gave up. (P8)

P11 tried to run a script given to them by a collaborator but could not get it to work.3 In the end, P11 simply sent the data and asked the collaborated to run the script on it and email back the result.

> We are trying to build this new structure, a triangle that is a bit weird. So what I do is, I try to design the pattern that I think is going to fold the way we expect and then I send it to him and he runs it, and he’s like “no.” (P11)

In summary, using computational tools requires an understanding of the entire software stack, but this kind of information is often

---

3 We helped fix the issue during the interview: the script would overwrite the original source file instead of creating a new file.
assumed in documentation and few online tutorials focus on such obscure dependency management. A computational medium should break the tight coupling between hardware and computational environments, and instead merge the computational environment with the code. This would allow the code/environment artefact to be self-contained and easily shared regardless of the underlying platform.

9.4.2.3 Computational Disempowerment

Paradoxically, the very tools that provide the scientists with the ability to push the boundaries of RNA origami also created a psycho-social working environment where they frequently felt disempowered. Participants were vocal about how their tools made them feel helpless, but mostly placed the blame on their own perceived lack of skills rather than the design of the technology.

So now I’m following this tutorial on how to use this other script and it doesn’t work and that’s it. I have no idea what to do. I don’t understand how it works, at all. (P11)

When a script reports an error, it is in the form of error codes or snippets of a foreign programming language. Errors in the molecular structure that make it impossible to be created are expressed as software errors and have to be somehow deciphered back into the realm of molecular biology.

This is one of my points of frustration. Because I have no idea what that [highlights terminal error] means […] I have tried to look at the script. And that’s about as far as I got. Because this means nothing to me. So if I go to line 2856 … 2856 is … print oped spool, okay yeah that’s what’s there [laughs]. (P10)

In fortunate situations, a more capable peer can assist, as when P11 explained how they would send data to a remote collaborator to run a script that would not work on their own computer. However, this is a fragile workaround as P11’s work became dependent on the collaborator’s time and goodwill.

Sometimes it is obvious that there are more efficient ways to go about a problem, but time and skill prohibit doing something about it.

I think this is one of those things where it’s definitely not the most efficient way of doing this, but it’d take me longer to find a more efficient way. (P10)

P8 explains how one of the scripts requires not having duplicates of the same sequences, but is a bit uncertain whether it is a biological or software limitation.
RNA wise it makes sense. But for some of these it doesn’t . . . I kind of like having duplicates. (P8)

Even though P8 would like to be able to change the script they don’t, instead they “hack the structure a bit” (P8).

The new computational possibilities are a blessing and a curse. They enable scientific breakthroughs that were previously inconceivable, but also create a dependence on machinery that the scientists are not trained to operate. When breakdowns happen—and they happen often—the scientists are largely left to their own fate, to much frustration.

9.4.3 The Computational Labbook Prototype

We produced a prototype through which to co-reflect with participants on their current and imagined future computational tools and activities. The findings presented above largely came from the co-creation of the prototype and its subsequent role as an artefact to talk about their current challenges and possible futures. In this section, we will describe the particular task that this prototype was developed for, how it implements the computational media principles suggested by previous work, and how its design addresses some of the three challenges faced by the scientists mentioned in the previous section.

9.4.3.1 Supporting the Designing of Macromolecules

Our prototype of a computational labbook[^4] is designed to support the design of RNA nanostructures, also referred to by the scientists as the “dry” part of the experimental process. This phase typically consists of some variation of these steps:

1. Producing a 2D textual representation of the target macromolecule, often by adapting an existing structure (from previous work or another lab member) with a molecule that folds in a desired way.

2. Checking whether the structure is physically capable of folding in the desired way by creating a 3D representation of it and looking for flaws.

3. Computing possible genetic sequences that might fold into the desired macromolecular structure and ranking them by viability.

4. Turning the 2D textual representation into a string-based representation that can then be send to a commercial company when ordering.

[^4]: See the accompanying video for an overview of the prototype.
Step two and three are largely interchangeable, and at least one researcher does not create a 3D representation at all. The “dry” part is then followed by the “wet” part in which researchers assemble, analyze, and validate their work in the lab space.

9.4.3.2 Redesigning the Task as a Computational Notebook

Besides providing functionality typically associated with ELNs, such as text editing and image uploading in a document, the computational labbook prototype lets the user add various computational instruments to the document by dragging and dropping from an instrument panel (see Figure 9.4).

The prototype addresses the steps of the dry phase in the following ways: To design a new structure (step 1), the scientist can create a new document and drag in an ASCII-based Pattern Editor instrument for the textual representation of the RNA structure. To preview the structure (step 2), a PDB Viewer instrument can be added, which generates an interactive 3D visualization of the structure in the document. Now, the Batch Revolver instrument can be added to compute multiple candidates of molecules (step 3) for a specific pattern scaffold by—broadly speaking—repetitively filling the scaffold and computing its energy properties. This is necessary as it is not feasible for researchers to try out every possible combination of nucleobases in their scaffolds which can contain hundreds of nucleobases. The resulting candidates can then be used for further analysis in other external

---

5 The Protein Data Bank file format.
6 Put simply, nucleobases are the four fundamental building blocks of DNA/RNA molecules.
tools. As the computation of candidates is computationally heavy, it is offloaded to a server which — once finished — sends the results back to the document.

The ability to offload heavy computation to a fast central server was (unsurprisingly) something that was well received by the participants.

*It works pretty well! [laughs] This is so fast. This is really good.* (P8)

Other instruments can be added to perform analyses on the structure, to perform various sequence operations (step 4) such as converting from RNA to DNA or reversing a sequence, or to give access to a shared repository of molecule designs.

*Having the repository is nice because it’s just click and drag and not a text document, stuff like that. But having it shared is also nice because you can make sure people are using the same motifs, like this mango one [a particular structure] has been through like five iterations.* (P10)

Multiple designs can be made in a document, and the computational instruments can be interleaved with written text, images, and even hand-drawn sketches if the notebook is opened on a tablet with a stylus. The unification of the different tools and scripts in one document was particularly highlighted by the participants.

*[whispers] Oh this is really convenient … usually I would have to like go in … make the modification, save the file, run the script on the terminal, wait for it to generate the file, then click that.* (P8)

*It would be nice to have the whole … everything from trace analysis to Revolver all on there. I think it would be nice to do … because it’s all amalgamated into one location. And it’s a well documented workflow rather than having … [clicks and shows folders of files] this.* (P11)

Using and discussing the prototype also spawned new ideas, e.g., the ability to chain the output of one script to the execution of another.

*This is pretty fast! Except you don’t get the summary, I would love to get these results but run another script on it.* (P8)

### 9.4.3.3 Adherence to the Principles of Computational Media

The prototype was developed to embody the four principles of computational media: shareability, distributability, malleability, and computability. We realized these principles partly by leveraging existing features of Webstrates and Codestrates and extending them with new
ones. Table 9.1 gives an overview over how the prototype practically implements these principles through specific functionality.

The principles are realized in the following manner: Notebooks are *shareable* as they are accessed through their URLs, they can be copied, and they support real-time collaborative editing and interaction. Nanostructures can be shared between notebooks through a repository. Also, functionality can be shared between notebooks using its built-in packaging system from Codestrates. While the participants did not collaborate closely on a day-to-day basis, they reflected on how the shareability could help in knowledge sharing.

*I think this is just so much nicer, just from an organizational point of view. Like if he’s working on these designs, instead of just sending me the text files and then I have to organize that in my own computer and yada yada, I can just look at his labbook and see the output, and see what he’s done.* (P10)

The prototype is web-based and can be accessed from any device with a web browser, and as a result it is *distributable* across a variety of devices (e.g., desktop computers, laptops, tablet, and phones) and operating systems (e.g., Windows, macOS, Android, and iOS). It supports distribution of interaction where, for example, the PDB Viewer can be shown on a tablet while the structure is edited on a laptop. Also, it supports distribution of computation. While the benefits of distributing computation was obvious to the participants, it was harder for them to imagine how to leverage distribution of interaction across devices in their day-to-day work.

The prototype supports *computability* as computations can be executed directly in the documents of the labbook without any setup, and it supports the execution of multiple programming languages. Compared to a conventional computational notebook, the prototype does not have code front-and-center. The code for the computational instruments can be accessed and modified from within the notebook, including the Perl and Python scripts developed in the lab. The built-in versioning support in Webstrates enables to revert to a working state if the notebook breaks. The notebook includes an instrument for creating custom scripts in Python, two dialects of Perl, Ruby, or Node.js JavaScript. The execution of the custom scripts is offloaded to a server. Hereby, the notebook supports *malleability*.

9.4.3.4 Technical Foundation

The prototype is realized using Webstrates [162] and its authoring environment, Codestrates [257]. Webstrates is a platform for creating computational media that follows a document-centric approach to software. Documents in Webstrates are identified by their URL and can also be shared using it. Every document is self-contained and
The design of the tools used by researchers play an important epistemological role in shaping what knowledge can be created. In his book *Changing Minds* [77], diSessa explains how the invention of modern algebra at the close of the 16th century suddenly made it possible...
for secondary school level students to grapple with mathematical proofs that were previously only accessible to university-educated mathematicians. A parallel can be seen in the tools for the origami method in biomolecular nanoscience: modeling nanostructures was initially done by hand, but this limited the research to only small, manageable sequences. For DNA origami, a group of researchers released a software packaged called caDNAo that provided a graphical user interface for designing 3D nanostructures [83]. All of a sudden, this branch of research became more accessible and more popular, to the point that “undergrad students can now even build these types of structures” (P3). For RNA origami, which the researchers in this study focus on, the tools are in-house developed scripts that compute suggestions of RNA structures. Because the script was designed to process the inputs sequentially, when it gets stuck in an infinite loop on a particular structure, all possible nanostructures that might have been “discovered” afterwards are rendered scientifically impossible.

Our findings highlight the dichotomy between computational tools designed as scripts and applications. Software as a script provides the nanoscientists with great expressiveness and flexibility to do research on the frontier of their discipline, but becomes unapproachable because of the competences required to create, execute, and maintain it. Software as an application is more accessible because it requires less computational literacy, but they provide only turn-key operations that calcify a particular scientific praxis and the types of research that can be done. We could teach all scientists to also be software engineers so they could make their own tools, or we could provide all labs with professional programmers to create user-friendly applications. These are both commendable, albeit unrealistic paths. A scalable solution is to develop a software model that straddles the divide of the dichotomy. This means questioning the application paradigm of the last 50 years and reinvigorate the efforts in creating computational media.

What design should computational media have? With our prototype, we applied four design principles derived from Klokmose et al. [162] and Rädle et al. [257] as a starting point: distributability, shareability, malleability, and computability. The rest of our discussion is structured around these principles, their relevance for the context of biomolecular nanoscience, and their limitations.

9.5.1 Distributability

Based on our observations and interviews, we can extend the concept of distributability for computational media by dividing it into three: distribution of documents, functionality, and computation. The distribution of documents is important because the participants need easy access across devices (laptop, desktop, tablet, phone) and spaces (office, home, laboratory, conferences). This need is already well-integrated
into the mental model of the researchers, and facilitated by file-sharing platforms and the Confluence system. The distribution of functionality refers to the orchestration of operations across devices. Currently, operations are bundled in applications which means that each device needs to install the same application for functionality to be distributed (i.e., to access spreadsheets stored in the cloud, every device needs to install a spreadsheet application). However, the participants did not engage in any multi-device activities (e.g., using the PDB Viewer on a tablet while using the Pattern Editor on a desktop computer) despite being provided with extra tablets by the lab. Previous research shows that—even if it is beneficial to use multiple devices to perform a task—users tend to use only one or at most two devices at a time [254]. Distribution of computation was something that made an immediate positive impact and was wholeheartedly embraced by the participants. Being able to offload heavy computations to a remote server was a crucial improvement over their current workflow, and is also a core focal point of many e-Science efforts. This, however, also necessitates server infrastructure that needs to be set up, be secure, should be scalable to a large number of users, and needs to be maintained.

9.5.2 Shareability

The participants work independently most of the time, so they did not see the need for real-time collaboration on a day-to-day basis. Instead, what the participants needed to be shareable was self-contained, computational environments. The difficulties they experienced trying to run scripts across devices make sharing their tools and results a huge hassle, either when when calling in the help of a more capable peer to debug an error, collaborating with a remote colleague, or networking at a conference. The problem of the tight coupling between computational environments and the hardware it runs on was similarly highlighted by Guo and Engler [121]. They built the CDE system, which combines code, data, and environment into software packages that could be transferred seamlessly between Linux machines. Our prototype facilitates shareability by storing and running documents/environments on a server that are accessed through URLs, which is a conceptual switch from the traditional pass-by-value of sharing to a pass-by-reference model. Beyond this, however, the prototype does not implement any functionality that supports collaboration. Research into groupware (see [120] for a historical overview) shows that when multiple people can access a document, questions such as “Who did that?” or “What happened since I was here last?” become pertinent [122]. We see additional challenges when the document is computational, with questions such as “Is something computing?” or “What computation is this a result of?”. These need to be addressed in
future designs, especially for scientific contexts where provenance of
data and computation are important to attribute authorship and settle
intellectual property disputes.

9.5.3 Malleability

The kind of software malleability we observed was not a constant
low-level tinkering with the code. Rather, it was similar to the mal-
leability of a house, where adjustments can be made when needed, but
which are often outsourced to skilled workers often such as carpenters
and plumbers. As such, the concept of malleability needs to take a
more collective view of software transformation, especially when the
computational media targets users who are not trained programmers
but rely on more capable peers. We implemented this by leveraging
the package system of the CodesTrates platform, which makes the
medium extensible through packages authored by others. This allows
users with limited technical skills to mold their environment, while
retaining easy access to the code should it be needed.

There is also the question of whether malleability is actually desir-
able, especially in the context of laboratory notebooks. While this did
not come up explicitly in our process, Oleksik et al. [239] document
how the ease with which ELNs are edited clashes with the “require-
ments for persistence and consistency of scientific records” and makes
collaborating with others more difficult because document structures
are no longer standardized. They suggest such systems could imple-
ment “fixity” to address the first concern: features that allow users
to freeze certain states of a document so it can be authentically repli-
cated. To facilitate collaboration, they recommend adding functionality
that can deal with multi-structured data, such as text mining, natural
language processing, and scheme integration.

9.5.4 Computability

Because the participants operate in a space where there is no estab-
lished software ecosystem, they opportunistically use scripts from
different sources, written in multiple languages. Therefore, supporting
computability in a computational medium should not just be about
the ability to execute code, but should also allow the agnostic mixing
of multiple programming languages in the same space. Our prototype
distributed code execution to containers on a remote server, which
makes it in principle possible to run any code. However, this simply
moves dependency management to a device out of the control of
the user, which does not scale very well without requiring the help
of experts to maintain those execution environments. With multiple,
heterogeneous components also comes the need for piping the output
of one into another — something participants requested. While our
current prototype does not support this, we have developed reactive data-flow pipeline mechanisms using Codestrates elsewhere [16].

Ideally, the day-to-day access to a medium that supports computability will improve the computational literacy of its users over time. There is evidence that this happened in the early days of the web through sites such as Myspace and Neopets. These social media helped users pick up HTML and CSS skills simply because the code was accessible and could be tinkered with [250]. Whether such learning-through-exposure extends to imperative languages such as JavaScript, Python, or Perl is unknown to us. What we do know is that creating the type of interactive components that we have developed for the prototype requires being able to read and write asynchronous event-based code, which is difficult to pick up by simply stumbling into an existing codebase.

9.6 conclusion

The tools used in research shape the science and the scientist, guiding what knowledge can be created and which competences are required to create it. In this paper, we explored the scripts and applications used in biomolecular nanoscience, and the concept of computational media as an alternative software model that sits in between them. We discussed how the computational culture of biomolecular nanoscience creates a dependency on digital tools, but does not have a tradition of training the scientists in developing, deploying, and maintaining them. We showed how, contrary to what most policy around the digitalization of science focuses on, the nanoscientists do not need help with learning how to code, but with managing the computational environments required to execute that code. As a result, the tools that make their research possible also are a source of feeling computational disempowerment on a daily basis.

Computational media — systems where users can author and execute code in the same perceptual space as other multimedia content — could provide biomolecular nanoscientists with the flexibility of scripts together with the accessibility of applications. During a two-year participatory design process with the Andersen Lab at the Aarhus University iNANO Center, we built a computational labbook prototype to explore this potential. We used the prototype to critically evaluated four design principles for computational media suggested by previous research: distributability, shareability, malleability, and computability. We find that distributing functionality across devices is not particularly relevant, but distributing (heavy) computation is crucial; that being able to share self-contained execution environments could play a small but critical role when collaborating and debugging code with more capable peers; that malleability is a collective
rather than individual need; and that computability should support the execution and coupling of multiple programming languages.

For the past 35 years, applications have been the dominant model for software. But the emergence and popularity of code-centric computational notebooks (e.g., Jupyter) and no-code computational documents (e.g., Notion) are exciting hints of a changing paradigm. Without a dominant design in place, what these computational media will look like is still an open challenge. Although our study is rooted in a very particular context, we hope it inspires a larger discussion around the paradigm shifting potential of this type of software.

ACKNOWLEDGMENTS

We would like to express our gratitude to the participating researchers from the Andersen Lab, in particular to Ebbe Sloth Andersen, and to the Interdisciplinary Nanoscience Center at Aarhus University. This work was funded by Aarhus Universitets Forskningsfond and Carlsbergfondet.
“IT LOOKS LIKE YOU DON’T AGREE”: IDIOSYNCRATIC PRACTICES AND PREFERENCES IN COLLABORATIVE WRITING*

IDA LARSEN-LEDET,† Aarhus University, Denmark
MARCEL BOROWSKI,† Aarhus University, Denmark

ABSTRACT

This paper addresses collaborative writing in academia. Recent research has indicated that while many tools for collaborative writing exist and continue to be developed, co-writers frequently employ workarounds and cumbersome substitutions to accommodate their writing approaches and collaborative needs. As part of a process to address these issues, we conducted a co-design study on collaborative academic writing with 18 participants. The paper details a three-stage co-design approach developed for this purpose. During this three-stage workshop series, the participants discussed needs, frustrations, and desires in their experiences with collaborative writing. These discussions revealed how participants’ different ways of practicing and experiencing collaborative writing entail contrasting needs that are difficult to balance. Based on an analysis of discussions and artifacts from the workshops, we argue that researchers and designers should aim to support diverse practices and propose a protocol for examining and drawing on the contradictions that arise from co-writers’ idiosyncratic preferences.

10.1 INTRODUCTION

The practice of collaborative writing is commonplace in the academic field: From undergraduate students collaborating on assignments and theses, through graduate students writing scientific papers, to professors and senior researchers who write, e.g., books as well as papers, to provide a few typical examples. As diverse as the co-writers carrying out collaborative writing processes together are the tools they bring together to support them in their endeavor. Beck [29] stresses

† Both authors contributed equally to this research.
the importance of addressing the dynamics of collaborative writing projects, along with the difficulty of anticipating particular dynamics. However, as recent research has indicated [72, 177, 242], the tools in use often do not fully support the idiosyncratic workflows of co-writers and push them towards workarounds and compromises to overcome limitations. The questions that we address in this paper are: What do co-writers experience as challenging in collaborative academic writing, and what steps can be taken to address those challenges?

While previous work has applied many approaches to investigate these issues (see Section 10.3), the implications they present for tool design are mostly based on the analyses and interpretations of researchers without direct involvement from co-writers. To fill this gap, we conducted a three-stage co-design study with 18 participants that consisted of a series of workshops. We worked together with academic writers (Master’s level and higher) and involved them directly in the ideation and design process for tools to support their writing practices. Our goal was to identify themes of what they liked and disliked in their collaborative writing experiences, to generate ideas for possible design solutions, and to, finally, use a high-fidelity research prototype to experience how their ideas would pan out. The writers who participated were used to different kinds of collaboration; both co-located and remote, and at different levels of synchronicity. In conducting the study, we found that individual idiosyncrasies result in contrasting needs and desires among co-writers. Our findings point to a more general problem, namely that these idiosyncratic ways of writing collaboratively are often hard, if not impossible, to accommodate in one solution [285]. Accordingly, our contributions are: (i) an identification of themes in the challenges experienced by academic co-writers; and (ii) a proposal to supplement design guidelines by framing the design problem of collaborative writing in terms of contrasting needs and desires. We propose a protocol to help designers and others approach and draw on these contrasts. Additionally, Section 10.3 thoroughly details the approach taken in order to allow others to draw inspiration from the study design.

Figure 10.1: Participants’ handwritten notes and sketches from the workshops.
10.2 RELATED WORK

This section, first, provides examples of the many topics addressed in guidelines for collaborative writing tools, in order to frame our argument that an approach is needed to help researchers and designers navigate and usefully apply these guidelines. Second, we also acknowledge existing work on ideas similar to those presented in this paper.

10.2.1 Guidelines for Collaborative Writing Tools

A number of authors have provided guidelines and recommendations for the design of collaborative writing environments, based on interviews [17, 72, 156, 224, 255, 261, 285, 305], questionnaires [224], observations [224], and/or logging or visualization of activity [242, 285]. One theme in these recommendations is the content and format of communication [17, 261]. Suggestions include, e.g., means for grabbing co-writers’ attention [242]. Related to this are recommendations to provide an overview [261], typically of a document’s revision history [72, 156, 255] or of other writers’ current activities [17, 305]. Considerations about this, among other things, include the granularity of what is recorded [156] and enabling identification of who did what [72, 242, 255, 305]. Recommendations also address planning and structuring the writing according to divisions of work in the writing [17], including various degrees of enforcement by the software. Yet another theme is support for multiple ways of writing, such as by facilitating regulation of co-writers’ access to individual parts of the writing [156, 305] and enabling different and potentially solitary activities that are part of the writing [255]. The general need to accommodate individual needs and preferences has also been acknowledged [72, 285, 305].

Building on literature, Neuwirth et al. [228] propose parameters to consider regarding communication and accessing and sharing content. These parameters are more open-ended than most guidelines but still include strong presumptions about the way the writing takes place.

10.2.2 Tools for Collaborative Writing

Collaborative writing tools have a long history and numerous features have been explored by authors during the past decades, covering a range of purposes. There are certain trends in this, which are reflected in our findings. We name relevant trends here, along with examples of features.

Features for communication and planning were among early developments: The tool Quilt [103] included ways to annotate and suggest changes in the text. A related theme is embedded information, such as highlighted background color in the text to indicate authorship [292] or “the age of a modification” by a background color fading over
time [293]. Quilt also enabled writers to track the status of the text; a feature also available in the tool Shared Books [181]. To support tracing and explanation of the development of the text, Neuwirth et al. [227] investigate diff-ing and annotation of edits, while others have shown how adding “meta-commentary” to documents could be used by co-writers to clarify edits [321] and how a “rational version control” could allow users to enrich their edits with explanations of the underlying rationale [309], features later recommended by Birnholtz and Ibara [34]. The early system AUGMENT [91] similarly included “abstract-like descriptions” of file contents. Shipman III and Hsieh [276] suggest that insights into co-authors’ motivations could also be gleaned from “replaying” the development of a document, as was possible in their Visual Knowledge Builder system. A recent implementation of such a feature is Somers’ Draftback extension for Google Docs [280]. Other exploration has involved utilizing the revision history to visualize the development of a document [170, 216, 281, 296]. Neuwirth et al. [227] note the granularity of such information to be relevant, e.g., due to the ways visual clutter can impact co-writers’ reading experience [157]. Somewhat related to this is the possibility for allowing writers to have different views on the text and the workspace [62, 264]. Finally, privacy has been addressed, such as with private messages [103] or ShrEdit’s [213] private windows [84]. Ignat et al. [137], e.g., proposed several privacy levels that include features like “anonymous changes” and “ghost operations” to hide edits from other co-writers until authors decide to commit them.

10.3 Co-design study

We conducted a three-stage series of co-design workshops over the course of five months. In this section, we first introduce our understanding of co-design and our motivation for employing the approach in this study. Next, we describe each of the three stages of workshops and how they connected to each other. Finally, the last subsections provide an overview of the participants and the data collected during the study, as well as how the data was analyzed.

10.3.1 Why Co-Design?

We use co-design to mean acts of joint creativity [270, 317] where we as researchers work together with end-users in a design process [43]. Taking outset in the Scandinavian tradition [11, 18], we aimed to empower co-writers to take part in shaping the tools they are using by actively participating in the design process. While this tradition emphasizes both co-creation and political agency, we focused on joint ideation and creation. We invited people with experience in collaborative academic writing to contribute and apply their perspective on
the topic, as experts of their practice [270]. In this study, it was not a case of the authors not possessing experience with the domain in question, seeing as a large part of our own work involves collaborative academic writing. Rather, the aim was to include a broader set of perspectives than merely those of the authors. The primary purpose for us was to enhance our understanding of what works and what does not work for academic authors writing collaboratively, while ending up with a design or a prototype was secondary and part of the process to increase the participants’ and our understanding of their preferences. The research project of which this study is part will, in the long run, contribute to improved support for collaborative academic writing — this study is one stepping stone among several carried out or planned.

Collaborative writing has frequently been studied using interviews [e.g., 4, 72, 156, 255], lab studies [e.g., 17, 35, 38], and in recent years also visualizations of writing activity [e.g., 176, 281, 304]. Other approaches include questionnaire surveys [29, 230], field experiments [62, 169], document analyses [70, 79, 315], ethnographic inquiries [69, 224], and auto-ethnographic reflections [e.g., 24, 240]. Finally, collaborative writing tools have been described and assessed in lab evaluations [e.g., 171, 296, 321] and deployment studies [3, 216, 285]. While these bodies of work all shed light on collaborative writing practices and outcomes, their implications for the design of collaborative writing tools are generally derived from analyses of the main findings, i.e. they are second-hand interpretations, and are rarely confirmed with participants. Our study is filling this gap by involving participants directly in ideation and design for collaborative writing.

Co-design in a way sits in-between methods such as field studies, that obtain accounts and observations of practice, and evaluations that obtain opinions on and assessments of technological artifacts [43]. We have been unable to identify previous research on collaborative writing using co-design.

10.3.2 The Workshops

The workshops followed a three-stage process with each consecutive stage building on the previous. We conducted five workshops in total (see Figure 10.2), two in stage one (W1a and W1b), one in stage two (W2), and two in stage three (W3a and W3b). The authors acted as the facilitators in all five workshops. The date and time for each workshop were based on an indication of availability from each participant, to ensure that as many as possible could join. Some participants were unable to attend workshops in all three stages but participated in the ones they were available for. The first and the last stage were each split into two workshops, to accommodate a large number of participants without overcrowding the workshops. We opted to keep
Figure 10.2: Timeline of the study, including the recruiting phase, conducted workshops, and the development phase of the prototype.

Figure 10.3: Overview of the stages and their outcomes. The preparation of Stage 3 was executed by the facilitators of the study.

the workshops short — each lasting two hours — seeing as participants volunteered to take time out of their working days or free time in the evenings to participate in the workshops without any monetary compensation or relief from duties [43]. We invited participants to attend a joint meal after each workshop to thank them for their time.

The three stages were characterized by divergence (Stage 1) and convergence (Stages 2 and 3). The first stage aimed to identify themes in collaborative writing through discussions and creativity exercises. The second stage built on these themes to generate ideas for possible design solutions on paper. Between the second and third stage, we prepared a software prototype implementing some of the ideas generated during the second stage. The prototype was used in the third stage of the study where participants could use and modify the prototype, to let them discuss something tangible.

By involving the same participants in several stages of ideation and design, we aimed for participants to experience their ideas evolve and come to life. Hence, bringing material from one workshop to use in the next was important. The project promised no direct benefit to participants; however, we were hoping for participants to still take something home from the workshops: By exchanging writing experiences with other participants, by getting insights into their own approach to writing collaboratively, and by being able to see their ideas manifest in a prototype, participants could reflect on and perhaps improve their own writing practice. In the following, we will describe the individual stages of workshops. An overview of the goals, employed methods, and outcomes of each stage is outlined in Figure 10.3.
STAGE 1: The aim of this stage was to identify themes in participants’ experiences with collaborative writing, as well as divergent thinking and ideation about how to address those themes. We wanted to help participants think beyond familiar tools and interfaces, to move away from suggesting feature improvements and instead envision freely what a desirable collaborative writing practice looked like, or could look like, for them. To accomplish this, we made use of brain writing [300] and a technique inspired by the notion of conceptual blending [96]. Conceptual blending is a theory of cognition that describes human understanding of concepts as arising from links between inputs from multiple “mental spaces.” To stimulate reflections and conversation, participants were paired up after an introduction to the facilitators and the study, and were asked to briefly discuss their most recent collaborative writing experiences. Then followed a group discussion where participants described and discussed “ups and downs” in these experiences as well as desires and wishes for collaborative writing tools. After this, participants were given a short introduction to the notion of conceptual blending and were then asked each to select one of the wishes (which had been noted on a board by a facilitator) to work with. Participants individually sketched diagrams of conceptual blends (see Figure 10.4 for an example), depicting their perception of their selected wish. They were asked to include a counterfactual mental space [96, ch. 11] in which their wish was not fulfilled. This was inspired by Pierce and Paulos’ [252] idea of counterfunctional things as a means to explore and design alternative experiences.

Based on their counterfactual mental spaces participants were asked to phrase a design challenge aiming to make the counterfactual wish a reality. For example, P11 who created the mental space diagram in Figure 10.4 phrased the challenge to design “some structure where everybody will work independently without discussing the structure or the content” since his actual wish was to help co-writers create a structure before writing. We provided instructions on the brain writing method [300], where participants wrote ideas on index cards (henceforth idea cards) and passed them around to each other in silence, using others’ idea cards for inspiration. To further stimulate ideation, participants were instructed to always include an image as inspiration for each new idea, by picking a random image from a pile on the table containing 100 New Metaphors: Thing 2 image cards [192] (see Figure 10.5a).

STAGE 2: In the second stage, the focus was on convergence and the refinement and elaboration of concrete ideas to support collaborative writing. As preparation for this stage, we grouped the idea cards created in the first stage into themes to be used in the Flip & Integrate exercise (described below).
Participants started by discussing, in pairs and then in plenary, what they considered essential for collaborative writing. The facilitator took notes about the participants’ discussion on a whiteboard. Thereafter, the participants and the facilitator jointly identified focal points in the whiteboard notes after which the participants formed groups of two to three participants—four groups in total—based on which focal point they would most like to work with.

In the group work, the groups first carried out an exercise that we termed *Flip & Integrate*. The exercise was inspired by the SIL method, or *Successive Integration of Problem Elements* method, in which ideas are integrated into a joint idea one by one [300]. Each group was supplied with a folder for each of the pre-identified themes, containing copies of the idea cards in that theme. In the *Flip & Integrate* exercise, the groups were asked to pick the theme they felt was most important to their chosen focal point. From that theme they then selected one idea and “flipped” it into a useful or realistic idea. In the first round of the exercise, the flipped idea was described in more detail as a design solution for the focal point. In later rounds, the current flipped idea was integrated into the design idea. When a new round started, participants picked the theme most important to their current design idea. The exercise was repeated, integrating more and more idea cards into the groups’ design ideas, for about 20 minutes. During the exercise, the groups were also asked to consider how to adapt their current design idea to three scenarios (see Appendix A.1) between each round.
10.3 CO-DESIGN STUDY

Second, we employed the Lotus Diagram exercise [219] (see Figure 10.5b). In this exercise, the groups had to reflect on eight concepts concerning their design idea. The eight concepts were derived from a previous study that focused on recent collaborate writing experiences of 11 of the participants (see Section 10.3.3 and Larsen-Ledet and Korsgaard [176]). The concepts were: Alignment; Distractions; Presentation of Self; Drafting; Yours, Mine, and Ours; Etiquette; Double-Level Language; and Local Expertise. Each group was provided with definitions of the concepts, to consult as needed. After the two exercises, each group presented their design idea to the other participants and discussed it with them.

Stage 3: The last stage of the study focused on giving the participants something more tangible than notes on paper. We developed a prototype in between the second and third stage (see Figure 10.2). The prototype was implemented using the Webstrates [162] and Codestrates [257] platforms including the latter’s package management [52]. Using these platforms allowed us to create an extensible and malleable prototype for participants to use as a sandbox to play with their ideas. While extensibility enabled participants to add or remove tools from the prototype while they were using it, malleability allowed them to re-program and modify the tools on a source code level directly within the platform.

The prototype included the following tools that were based on participants’ ideas: Edit Display; Edit Overview; Edit Overlay; “Where am I needed?”; Paragraph State; Paragraph Voting; Comments; Meta Notes; Paragraph Locking; and Inspiration Prompt. To give participants an overview, we created a pamphlet describing how to access and use the prototype and summarizing which tools were available—including scans of notes of the previous workshops to indicate the tools’ originating ideas. Giving participants a nice-looking pamphlet showcasing implementations of their ideas was part of our effort to show participants that their contributions were valued and taken seriously. After introducing the prototype and its tools to the participants,
they were divided into groups of two to three participants—three groups per workshop, six groups in total. Each group was provided with an electronic document and were first asked to explore the prototype and the provided tools (participants used their own laptops for this; see Figure 10.5c).

Next, each group selected one of three scenarios (see Appendix A.1) and started (1) choosing which tools to use for their ideal editor for their scenario, (2) modifying tools to better fit the scenario and their own needs, (3) creating storyboards for modifications and ideas that they were unable to implement within the allotted time, and (4) filling out a questionnaire about the features and the malleability of the prototype (as part of a separate research focus of one of the facilitators). Halfway through the exercise, we introduced a disruption for every scenario, that changed the task slightly (see Appendix A.1). During the second half, the participants continued to further adjust their editor to this change. Finally, the participants presented their results and discussed the topics of the questionnaire.

10.3.3 Participants

18 people participated in the co-design process (see Table 10.1). Five participants described their gender as female, 13 as male. Seven different European nationalities were represented, although most (11) participants were Danish. Participants had different native languages; thus, all workshops took place in English. Participants’ ages fell within the span 18–55 with the majority (10) being between 26 and 35 years of age. Participants’ occupations fell within the following categories: Master’s students (2), Ph.D. students (5), postdoctoral researchers (3), assistant professors (2), associate professors (2), recently graduated Ph.D. (1), and software engineers recently graduated from their master’s (3).

Participants were recruited with a focus on experience with collaborative academic writing. Participants’ writing experiences included, e.g., writing a Master’s thesis in groups, writing academic papers, and writing books. Some of these writing projects had been undertaken mainly co-located while others had been remote or mixed. The writers likewise applied different strategies in terms of the level of simultaneity of writing [177, 255]. Since participants needed to attend multiple workshops in person, we only recruited people from our own university (the only university in the region). We placed posters and flyers in departments in all of the university’s faculties—each slightly re-phrased to explicitly address the community that the poster was placed within, in the hopes that this would increase people’s motivation to sign up. Some departments circulated a recruitment email or shared a Facebook post. Additionally, some of our colleagues with students, for whom experiencing a co-design process could be
relevant, informed their students of our study and encouraged them to participate. We also invited participants from a previous study on the same topic [176, 177], of whom 11 participants also participated in the present study. All of the new participants had current or recent experiences with collaborative academic writing, whereas some of the returning participants had finished their studies and were no longer doing collaborative academic writing. Despite the efforts to reach a diverse range of academic communities, six of the seven new participants, as well as all 11 returning participants, were studying or working in a technology- and/or design-related field. Most of the participants knew at least one of the other participants in advance. Only one had never met any, and one knew a couple by name but not personally.

### 10.3.4 Data Collection and Analysis

During the co-design study, we recorded video and audio, and took photos. We also collected artifacts made by the participants during the workshops: notes, mental spaces, and idea cards (Stage 1); notes and lotus diagrams (Stage 2); storyboards, a questionnaire with each group’s reflections, and the documents participants had used in the prototype as well as their changes to the code (Stage 3). Each participant also filled out an anonymized demographic questionnaire in the first workshop they participated in. The study was approved by Aarhus University’s Research Ethics Committee and participants have been informed and have given full written consent per European Research Council ethical guidelines.

For the analysis, all artifacts created by the participants, i.e. notes, sketches, filled-out lotus diagrams, etc., were scanned. For all plenary discussions and presentations, the audio from the video recordings was transcribed by one of the authors. Both authors did individual inductive coding [32, 90] of the transcripts and artifacts from the first stage, coding with a focus on needs and problems in participants’ current writing experiences. The authors’ individual sets of codes were consolidated and a joint set of codes was agreed upon. The transcripts and artifacts from the second and third stages were coded.

|          | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | P10 | P11 | P12 | P13 | P14 | P15 | P16 | P17 | P18 | N  |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| W1a      | ×  |    | ×  | ×  | ×  | ×  | ×  | ×  | ×  | ×  | ×  |    |    |    |    |    |    | 8  |
| W1b      | ×  |    | ×  |    | ×  | ×  | ×  | ×  | ×  | ×  | ×  |    |    |    |    |    |    | 8  |
| W2a      | ×  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | ×  | 11 |
| W3a      | ×  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | ×  | 7  |
| W3b      | ×  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | ×  | 6  |

Table 10.1: Overview of the participants’ attendance in the five workshops.
against the joint set of codes. Based on the coding of all three stages, we identified five high-level themes which are discussed below.

10.4 FINDINGS

The findings are divided into five themes: First, we discuss participants’ ideas for supporting the alignment of individual contributions. Second, closely connected with this follows a discussion of planning and how to preserve an overview of past, present, and future activities throughout the writing project. Third, we outline participants’ reflections on communication, which can be divided into presentation, type, and purpose. Fourth, we present ideas about different views or modes for supporting different activities in the writing process. Finally, we discuss the spectrum between joint and separate writing, in particular addressing participants’ perspectives on managing boundaries. After the five themes, we end the section by drawing up the contrasting needs and desires in participants’ discussions and ideas.

We refer to individual participants as P₁–P₁₈, to groups from the second stage as G₂₁–G₂₄, and to groups from the third stage as G₃₁–G₃₆. In the second and third stages, we refer to groups instead of individual participants when a statement or idea cannot be clearly attributed to a specific participant (such as is the case with, e.g., the notes and storyboards).

10.4.1 Alignment

Not surprisingly, the alignment of work contributed by multiple individuals to a common product was brought up by several participants, and was continuously addressed during the workshops. In this section, we use alignment to refer to aligning the text produced by writers [176], be it the content or the presentation of it. Planning and division of work — and hence alignment of expectations and of the work carried out — will be addressed in the next section.

For one, alignment entails coordinating the work of multiple actors towards a (more or less) shared goal. As P₁₆ expressed it, “when you sort of delegate stuff […] the problem tends to be that then you write some arguments that don’t follow each other.” Participants suggested solutions explicitly for this task in the form of automatic or semi-automatic detection of incompatible arguments, as exemplified by G₂.3’s design (a joking reference to the unpopular Clippy character [310] from Microsoft Office), which the authors refer to as “the Mediating Clippy,” shown in Figure 10.1a. The group clarified that the idea was not for the automatic tool to fix misalignment but for it to instigate communication:
If you would say: “What is the main message of this paper?” and if you would ask every person that, you could say: “Oh, it looks like you don’t agree!” [...] And then you could actually coordinate and communicate about these things and agree on them, and align your points. (P7)

Participants were also concerned with coherence in the writing style, or with how to “align the outward presentation” as G2.1 put it. This involves, according to participants, using consistent terminology and creating a text that “reads as one” (P9). This concern, which is likely particular to collaborative writing, played into a more general concern for producing well-written text. Altogether, participants’ ideas served to develop a general notion of how features for language support could be augmented to support collaborative writing. For example, G2.1 played with the idea of an auto-complete feature that would suggest words or phrases based on a database of previous writing (e.g. the authors’ own writings or papers from target conferences). This group wrote in their notes about the idea that “[a]utocomplete can create a group identity” (G2.1). G2.4 came up with a tool that would enforce a style defined by the writers, or by a writer in charge. As opposed to the Mediating Clippy, this tool would relentlessly require writers to follow the style guide. The group’s motivation was that it would allow co-writers to focus on other parts of the work, “so that once you start writing you won’t waste time arguing about different styles” (P8). A participant from a different group argued that such a system would be cumbersome to work with, playing into a frequently encountered dichotomy between individual control and the convenience that automation or software-controlled procedures could provide.

Finally, G2.3 suggested that alignment is connected with joint accountability, as they described that by explicitly approving a piece of text written by a co-writer, one takes on part of the responsibility for that piece of text (construed as a positive thing, contrary to findings of Wang et al. [305]). In addition to clear communication of a joint message, achieving alignment can thus also be about accountability and the individual’s and the group’s relationship to the text. P7 argued that a mechanism requiring co-writers to explicitly approve text passages for those passages to be included would “[force] everyone to actually look through the paper,” ensuring that everyone follows up on their commitment.

10.4.2 Planning and Overview

According to participants, planning serves to structure the text as well as the work that goes into creating it. At the beginning of a writing project, participants would try to align expectations among co-authors by outlining the structure of the text, agreeing on a writing style, and
defining procedures. Many of them mentioned creating bullet points for each section as a way to ensure that meaning is not lost in the writing process (see Figure 10.1b). A suggested feature was a way to annotate sections with such bullet points without having them appear directly in the content text (P14).

While these steps create an overview of future work at the beginning of a project, this overview further needs to be preserved throughout the writing. Tools like version control (G2.4) or indications of what state individual sections or paragraphs are in (G2.3) (see also Section 10.4.3) were among participants’ ideas for facilitating this. Version control allows co-writers to see how the text has evolved, and manual tagging of the state of a paragraph enables an overview of how much still needs to be done in a document. P1 expressed it this way:

Traces of use. What has changed to a text over time? Who made the changes? At what point were changes made? Being able to check the traces in the document. (Idea card of P1)

A couple of groups suggested ways of tying writers’ identities to their contributions, as part of providing an overview. G2.2, e.g., suggested that “[e]ach writer [should] have a colour that the text written is marked by” (also proposed by Wang et al. [305]) and that “[e]ach paragraph written [should be] annotated with the authors profile picture.” While G2.2’s approach focuses on the who, G2.3 made the note: “from who wrote → to status of it,” suggesting to move focus away from the individual to the jointly written text. G2.3 furthermore described reactive writing [194] as a benefit to live co-editing, noting how co-writers can “complete each other’s sentences” (G2.3’s notes), enabling them to keep up-to-date in real-time.

10.4.3 Communication

As mentioned, participants’ thoughts on communication mainly regard three aspects: presentation, type, and content. According to participants, the presentation of communication, e.g., how comments or suggestions are displayed, is significant for their collaborative writing experience. They brought up that while platforms often support comments, suggestions, and tracking changes, these features have no differentiation in granularity and can quickly clutter the interface. For example, P16 mentioned that all changes in Microsoft Word’s track changes feature look the same, making it hard to distinguish small fixes of grammar errors or typos from changes to the overall meaning of a paragraph of text:

All changes look the same and [have] the same space on the interface. So […] an obvious error that just needs to change has the same space as a […] discussion of the coherence. (P16)
Participants also discussed different types of communication, such as annotations and notifications. These discussions sometimes included the role of automation in communication. For example, P12 mentioned that he would like his “advisor to be forced to see whenever [he] make[s] a change.” In response to this, P3, who has supervised several students, replied that it would be better to get “a notification once something is finished” as “plenty of notifications all the time is actually pretty stressful.” This relates to the purpose of communication: In addition to communicating about plans or current activities, participants also stressed the ability to communicate their intentions, such as what they were aiming to achieve with a certain paragraph. They described how it can be problematic when someone loses track of the text’s connecting thread while editing:

“So I’ve often had the discussion of “what is the goal of a certain paragraph” or something. And then the goal stays pretty constant throughout the writing, but the text changes. […] And sometimes I feel like, if you have co-authors they will sometimes forget the goal of a specific paragraph or subsection. (P12)

Making such goals or intentions explicit in the document were thought by participants to improve the coherence of the text, and hence support alignment. One suggestion was to allow “metatext,” as P16 called it (see also, e.g., Rimmershaw [261] and Engelbart [91]). The bullet points mentioned in Section 10.4.2 served a similar purpose. Participants further discussed automatic generation of such metatext. Communication of intention could be supplemented by communicating the state of different parts of the text (as was possible in, e.g., Shared Books [181]). P1, for instance, suggested the phases: “In progress”, “Could you look at it? Suggestions”, “Don’t look at this yet”, “Don’t touch. Working on”, and “Finalized”.

10.4.4 Different Views

Participants also addressed the different activities involved in (collaborative) writing. A recurring theme was having different views or modes for different tasks (see also, e.g., the Col•laboració system [62, 264]). P14, for instance, differentiated between what he called “a focused” writing mode and a review mode with a more elaborate set of features for annotating, providing suggestions, and discussing changes, along with other forms of communication and revision:

“So, in the reviewing process I will have annotations and the ability to suggest changes and stuff. But in the writing activity […] you wouldn’t have those tools available. And they also wouldn’t clutter the interface. (P14)
Another topic was how different views of the text impact the writing experience. For example, P4 argued that “[a] txt file can also help you [to] sharpen your argument, because you look away from everything else and just [work on] the exact wording of that specific sentence or passage.” And P3 brought up how clutter can make it hard to switch to a reader’s perspective (see also Kim and Eklundh [157]) — this contrasts with the feature-rich reviewing mode suggested by G2.2:

&S]wapping into reader mode [is] sometimes difficult if you have a text that is full of comments. Sometimes even exporting it to a different template or even a PDF version […] you start seeing all the spelling errors and things like that. So […] something that could help you become a reader rather than a writer of the text. (P3)

In addition to enabling different views on the document, ideas for tackling this issue also included starting off in a “bare-bones” editor with typing and deleting as the only available options, allowing authors to add necessary features as needed (suggested by G2.2. The idea is reminiscent of, e.g., CoDESK [291]).

Beyond the user interface, the desire for different views also applied to the representation of text. While some participants preferred a clear separation of markup and content as it is found, for example, in LaTeX, others preferred a WYSIWYG (“What You See Is What You Get”) interface, as found in, e.g., Microsoft Word or Google Docs. For some participants, the choice was based on their experience of the tools, such as P7 feeling more in control when using LaTeX’s automatic placement of figures than when using the manual placement in Microsoft Word. For other participants, deciding on the tool was not so much a question of choice, but of theirs or co-writers’ familiarity with certain tools, or of organizational regulations imposed on some co-writers effectively forcing everyone to use a particular tool. Optimally, both types of views on the text could be available at the same time, as P17 proposed in his thoughts on “workspaces”:

But we also were talking about having different workspaces, so if Ida, for instance, [prefers] the LaTeX-way of doing things, with markup and such, she could do that. And if I want WYSIWYG I could have that. (P17)

10.4.5 Joint and Separate Writing

Participants also discussed benefits and downsides to different forms of writing together: Simultaneous editing in a document that is updated live was seen by some as making it easy to contribute to each other’s work (see also Section 10.4.2). Two groups thought of ways to tag co-writers in order to request help or feedback on a
specific passage of the text. G3.3 created a storyboard for this feature (see Figure 10.1c) and G3.2 implemented a minimal working version of it during the last workshop. Some participants went even further and suggested ways to take over the viewport of other co-writers in a shared document and bring them to the paragraph where they are needed:

> And we thought it would be really nice to actually, you know, request people’s attention to a certain spot. (P18)

> Overrule others’ workspaces to focus on certain sections or work processes. Authors can lock other writers screens to their view, or freeze a section that people are forced to look at. (Lotus diagram notes from G2.2)

Others noted how live edits made by co-writers to one’s work could be immensely distracting or frustrating. As can be seen from this exchange, some likewise found it difficult to catch onto or respect such boundaries:

P6: I hate when somebody is editing the sentence that I’m writing, in the moment.

P7: Oh, I do that!

P6: (laughing) That’s super annoying!

P7: But if you see a typo …

P6: Yeah, exactly, but I’m like: “I saw that as well!”

Another contrasting desire that participants discussed was the ability to focus in an undisturbed space (resembling other recent findings [177]), which was seen as an advantage to not writing simultaneously in the same editor. This also related to distractions that do not stem from direct interaction between co-writers: P6 described how she would lose track of her position in a shared live document when the text she was writing got “pushed down the page” due to co-writers pasting in text further up in the document.

Another advantage of not writing simultaneously in the same editor that some participants mentioned was being able to control when text is shared. Participants often referred to text as being “ready” or not. For some, it was a question of not wanting to review someone else’s writing until it was coherent, as mentioned by P3 in Section 10.4.3. Others named motivations having to do with presentation of self [114] and not wanting to share writings until they were no longer “bad text” (P8). Some also used phrases like: “someone looking over your shoulder” (P2) or “Big Brother” (P9) to describe live co-editing, explaining a desire to work on text without others being able to “watch” them. These sentiments echo other findings in recent research on collaborative writing [176, 305].
Participants came up with several ideas for managing boundaries in the writing: “Locking” sections or otherwise controlling collaborators’ (level of) access to text (e.g., G2.4) exemplify ideas that rely on enforcement by the software. Other ideas relied on the etiquette and discretion [176] of co-writers by adding means for writers to express how they would currently like others to treat their text, for example, by putting the text in certain visually apparent states, as described in Section 10.4.3.

An issue that was not tied to a particular mode of collaborative editing was ownership and control of text. G2.4’s idea centered around managing co-writers’ access to particular sections, with the explicit aim of preventing co-writers from “trash-[ing]” each other’s work. This included an idea that writers would be notified if others “visit/review/edit” their sections. G2.3 presented an alternative in which co-writers would vote on changes. Modifications to this idea included being forced to suggest alternative text before being allowed to delete any. Both of these groups’ ideas were construed as ways to avoid edit wars — to align on the text.

10.5 IT LOOKS LIKE THEY DON’T AGREE — USING TRADE-OFFS CONSTRUCTIVELY

In the sections above we have described alignment of the writing, and the role that planning and maintaining an overview play in this, as well as the roles of communication and the significance of being able to communicate intentions. We have also described participants’ ideas for supporting different activities and for reconciling different preferences. Finally, we have described the balancing of writing together and managing boundaries. In this section, we discuss contrasting observations of what is needed in design for collaborative writing. Many of the participants’ reflections and ideas had clear roots in tools they knew (of), and many of them proposed things that have previously been addressed in the field of CSCW, such as the ability to locate collaborators [84, 123, 124]. What we want to highlight are the contrasts that are present in some of those reflections and ideas. These are contrasts between the needs and preferences of different co-writers, as well as between individual writers’ personal needs and preferences.

10.5.1 Contrasts

calling attention and controlling disruptions: On the one hand, participants wanted as much information as possible. They particularly wanted to be able to share information, and to some extent also to receive it. In participants’ discussions of notifications we noticed two contrasting perspectives: notifications as a service for the sender, and as a service for the recipient. In the first case, notifications
were envisioned as a means for requesting someone’s presence or feedback. In the second, they were seen as a way to be made aware, e.g. that a piece of text is now coherent enough for feedback, or that a co-writer is making modifications to one’s writing. See, for instance, the discussion among P3 and P12 in Section 10.4.3: For P12, notifications would be a way to keep up to speed with what is going on in a document, while P3 found this to be unnecessarily distracting and saw notifications from co-writers as a means to know when going over the text would actually be useful. This contrast, among other things, speaks to the granularity of information: Making minute changes apparent can serve to give an overall picture of activity in the document, but such a detailed overview can make it hard to trace the development of overall arguments [156]. Neuwirth et al. [228] address this in terms of a solicit/subscribe setup, and in a discussion on calling attention to sections based on how much they have been edited, Olson et al. [242] caution that notifications should be subscribed to at the discretion of writers. However, none of them address how to balance the communication needs of one co-writer with those of another.

**information availability and tidiness:** While participants discussed ways of enhancing the writing environment with embedded information, such as visual indications of the state of a piece of text, they were also concerned about clutter. For example, using annotations or formatting to support boundary management around specific paragraphs [176] contrasts with the need for a clean reading space for going over the text. Allowing co-writers to enter such a clean state would make the embedded communication unavailable to them, and its value for enabling boundary management (without enforcing it) would be lost — unless, of course, co-writers are instead notified about text they should leave alone, or manually check for this, both of which would interfere with the reading experience in other ways and would hinder easy switching from reading to commenting or editing. Similarly, having different views, e.g., a drafting and a reviewing mode as suggested by participants, could impede communication if, for instance, a commenting feature is only available in one mode. Posner and Baecker [255] include easy switching between activities as a requirement, but do not address potential problems in considering individual activities as separate. Olson et al. [242] disagree with Posner and Baecker’s view, but do not offer clarification beyond recommending that the tool be “unstructured.”

**privacy and availability:** On the one hand, participants described how having a sense of what co-writers are doing in a particular moment can be helpful [194, 305], but they also wanted to be able to withdraw [176] in order to focus or to have a sense of privacy. Support for private writing has been suggested before [84, 305], but the
dilemma it poses alongside reactive writing has yet to be addressed. Wang et al. [305] propose augmenting private windows by displaying that a person is active, to avert assumptions that the person is not working. This, however, would not improve the possibility for reactive writing [194]. Posner and Baecker [255] argue that it should be possible to segment a document while maintaining connections with the entire document, but do not go into further detail about how to accomplish this.

Furthermore, this previous work only addresses privacy in the moment of editing. But we also noted diverging needs with respect to privacy over time: Participants saw a benefit to being able to trace from whom different ideas in the text originated, among other things in order to be able to align their arguments or seek explanation from each other. Being able to trace who wrote what clashes with the desire for privacy: Either the revision history of text written in private is kept unavailable, or part of the privacy of writing in a separate space is lost. G2.3, in fact, questioned the need to know who wrote what and suggested instead to focus on the state of the text. Several works mention the ability to distinguish who wrote what [72, 242, 305], but none of them address the place of privacy as regards what has happened. Although Wang et al. discuss balancing privacy and awareness they neglect privacy in their discussion of authorship indications.

**Sharing and Boundaries:** The notion of having the division of work manifest in the tool, for example by enabling writers to “lock” passages of text they are working on to keep others from interfering, contrasts with the aim to produce a text that “reads as one”; a contrast that was also voiced by a participant in response to the idea of locking text. Previous work has addressed support for differentiated levels of access to a document [103, 255], but although they also describe features supporting alignment (e.g., planning [255], status tracking [103]) they do not address the interplay between these sets of features, such as how differentiated access may negatively impact alignment.

**Automation and Manual Control:** Many of participants’ ideas involved automation to varying degrees. On the one hand, they envisioned how software could help them align their writing style or manage boundaries, as discussed above. On the other hand, they also described needing to be able to fluently adjust in particular situations. For example, participants discussed having a “minimum threshold” for the text locking described above, where everyone is always allowed to make minor edits. It did not become clear, however, exactly when something counts as a minor edit. We find that such a definition is, in fact, not possible in any useful way, as it would likely change depending on the circumstances. Allowing co-writers to declare this for each project, similar to G2.4’s idea for enforcement of writing
style, may introduce a kind of flexibility but would take away the moment-to-moment flexibility that the solution was supposed to introduce. Olson et al. [242] are critical of explicit declaration on the part of co-writers. They argue instead for a simple tool that can be used flexibly but do not go into further detail about what such a tool would look like.

Many existing guidelines are well-founded in empirical work and provide useful advice on how to design certain aspects of collaborative writing tools. However, by proposing standalone guidelines, the authors gloss over complexities that must be dealt with. Individual guidelines stand alone by pointing to particular recognizable aspects, such as desiring a clean reading experience or needing to know what co-writers are working on, leaving out the connections to other aspects. Hence, guidelines provide a reduced perspective that leaves out contrasts and results in a loss of nuances. By narrowing the focus to predetermined elements of a generically conceptualized writing situation, they fail to help researchers or designers explore what “could be,” given the particular characteristics of the people and context—i.e., they fail to serve an epistemic [229] function. Taking an epistemically generative [111] approach would entail leaving room for the particularities of each case rather than imposing a predetermined conceptualization. We propose that an epistemically generative layer may be added by framing the problem of designing for collaborative writing in terms of contrasts, like the ones we have outlined above. Contrasts can be seen as a resource for questioning assumptions [92] in order to press on from immediate possibilities for solutions that are anchored in what is recognizable from previous cases. The examination of these contrasts should instead be anchored in the particular case being addressed. Considering how to make space for a given contrast may result in a design that reconfigures the practice, in turn providing a new anchor for future solutions. With this paper, we intend to push for the elaboration of this framing. We expect that it will be helpful not only when studying and designing for collaborative writing but also with other kinds of collaborative work, not least if coupled with an iterative process directly involving users.

10.5.2 Trade-Off Protocol

As an aid in addressing contrasts, we propose the following protocol for framing design discussions using the contrasts as spectrums along which to discuss challenges. The protocol can be used both for ideation and for evaluating systems:

1. Select an aspect to work with (such as one of the five challenging aspects identified in the previous sections).
2. Identify (current or potential) mechanisms supporting that aspect.

3. For each mechanism, identify relevant contrasts and apply the following steps for each contrast:
   a) Using the two sides of the contrast as ends on a spectrum, note where the mechanism is on that spectrum.
   b) Consider (an) alternative design(s) that are more towards the opposite end of the spectrum.
   c) Consider what is lost and gained in the alternative design(s).

As an example, one might consider planning and overview and discuss G3.1’s metatext mechanism with bullet points hovering next to the text (see Figure 10.1b). One of the relevant contrasts to be discussed is information availability and tidiness. This mechanism is closer to information availability so the task would be to think of a design providing tidiness. This could for example be a version where the bullet points are shown only when the section heading is moused over. With this design, writers lose the chance of serendipitously noticing information they were not expecting. But they gain the ability to control when they are viewing metatext.

The protocol is meant to help designers and researchers draw creatively on contrasts. It is intended as a guide for reflection on trade-offs and does not prescribe one side of a contrast as favorable. In some cases, it may be desirable to strike a balance between contrasts by favoring one side in some mechanisms and another in others; while sometimes this may not be the most suitable approach. The protocol should be seen as a supplement to existing guidelines that may be used to nuance their application.

The hardest problem to address is that of co-writers’ different preferences and intentions. Involving co-writers in co-design can help draw out contrasts, but co-design participants should not only participate in identifying the contrasts: Participants should also preferably take part in applying the procedure using those contrasts. This way they will be made to articulate their practice as well as discuss and question it. Thereby, the procedure could potentially help co-design participants negotiate and (re-)configure their practice.

10.6 DISCUSSION

In arguing that contrasts must be explicitly addressed, we are not saying that we should design our way out of compromises in collaboration. With the benefit of involving multiple people in a piece of work comes naturally a necessity to compromise. Others have similarly addressed how certain aspects of a writing collaboration are or should
be negotiated, such as the order of appearance of authors’ names [86, 98]. However, compromises should be among people, not between people and computers. And software should not force unnecessary compromises between people.

Some of the dilemmas faced by participants in part come down to figuring out a joint practice and making it work — after all, some things are “solved outside the software” (P3) [255]. But as our participants — as well as our own collaboration on this paper — have taught us, shaping a joint practice is not an easy task. Designers should consider how to support this difficult task, and how to not inadvertently obstruct it.

Tammaro et al. [285] suggest that personalizable, or similarly flexible, groupware may be the only way to meet the needs of diverse groups. Advances in malleable software [59, 162, 260] make this worth exploring further. Malleable tools could enable different views, different representations of text, and adaptability over time — all things that were discussed by the participants. But while malleability can provide more flexible control over a tool on the technical side, this control is not necessarily aligned with the feeling of being in control on the user’s end. Some users do not feel in control when they can mold every aspect of a tool to their exact needs, they rather feel in control when there is no need to mold the interface at all [197]. Another way to provide writers with flexibility and control is to design for multiplicity [40, 44]. Rather than adaptations within tools, focus should then be on the potential to shape the artifact ecology [44, 144] by integrating new devices and software [45]. Such an approach would be in line with what Larsen-Ledet et al. [177] suggest. The open-ended nature of our proposed trade-off protocol is helpful in either approach.

10.6.1 Beyond Collaborative Writing

As we have noted in Section 10.2 and while presenting the findings above, many of our participants’ ideas and concerns are recognizable when compared with existing literature on collaborative writing. Some of them have also been addressed in literature on topics beyond collaborative writing, e.g., reading and group work, including flexibly annotating documents [275] (including specific guidelines [209]), coordinating and communicating among individuals in group work [215], calling attention to specific parts of a text [245], or transitioning between private and joint group work activities [215]. The fact that much of this work, while not directly addressing collaborative writing, describes activities relevant to the writing process further supports the assertion that flexible groupware and/or an ecological perspective is required, as suggested above.
10.6.2 Reflections on Co-Design

Whereas previous work has described practices and tool use in collaborative writing [e.g., 156, 177, 305], taking a co-design approach with people unfamiliar with research on collaborative writing has, in addition to confirming much of what has been proposed and discussed in previous work, provided a picture of the tensions among those things. While these tensions could possibly have been analytically teased out from previous work, co-design’s emphasis on exchanging perspectives brought them to light naturally by allowing the participating writers as well as ourselves to notice and jointly reflect on contrasts and compromises.

Exploring participants’ practices through ideation and design exercises served to bring out nuances to the experiences and preferences they described to us. The group discussions allowed participants to challenge each other’s preferences and opinions, which in particular contributed to the perspective on contrasts.

We chose to let all the artifacts created by the participants during the workshops be available to all participants throughout the three stages. Encouraging participants to build on each other’s ideas served to open alternative avenues for discussion and speculation and integrate more people’s perspectives in the ideas developed. Participants’ responses to each other’s ideas, both in the form of feedback and questions and in the form of new or modified concepts and designs, complemented the more opinion-centered discussions and further highlighted the contrasts. Although many of our participants’ ideas are recognizable in terms of existing work on collaborative writing, our participants have surely contributed novel ideas and angles that have helped us understand their perspectives as academic co-writers.

We see the three-stage co-design process presented in this paper as a viable approach to examining contrasts within communities of practice. Drawing on our experiences, we would recommend that if conducting a workshop similar to our third stage, with a pre-made prototype, participants be given more time to familiarize themselves with the prototype; if possible, by extending the workshop to a full day. We also recommend being careful to ensure that all participants find their practice reflected in such a prototype (see below).

10.6.3 Limitations and Future Work

The prototype design leaned towards a WYSIWYG style which naturally influenced what participants focused on in the third stage, and some participants may not have seen their practices reflected in the prototype as much as others. An approach to amend this in future work could be to augment the prototype in close cooperation with individual participants and deploy it with them for a longer duration. This would
enable long-term development and let participants reflect on their use of the prototype for actual writing tasks over a longer period, generating further insight into the needs that arise for them in their collaborative writing practice. We are weighing this against potential difficulties of this approach, some of which are evident in our own findings: People need advanced tools that they are familiar with \cite{72, 156, 215} in order to do their work, as do their collaborators.

Another point to be made is that all participants of our study had at some point worked or studied at the same university and, further, in mostly similar fields. While involving them broadened our horizon beyond our own individual perspectives, the findings presented still reflect a culturally narrow group. For this study, however, that has served to demonstrate that even within such a narrow group there were idiosyncrasies and contrasts in and among the practices represented. The aim of this paper has been to raise awareness about these contrasts and the importance of designing for different practices, even when focusing on a particular community of practice, in this case academic writers with a background in design or technology.

Finally, we reflect on the mindset that this work was carried out with. By inviting academic writers to act as designers rather than informants, we wanted to expand the space given to their voice as regards tools for collaborative writing. But we have made decisions without the participants that they could have been involved in. For instance, we could have let the participants select the features to be implemented in the prototype but opted to spend our limited time with participants on other activities. Participants’ agency was restricted to what topics they wanted to address and what they chose to prioritize during ideation.

With the writing of this paper, the agency has fully shifted from 20 people to the two authors only. Light \cite{183} articulates what she calls “the politics of describing others” and problematizes the typical approach to describing participatory design projects, in which the researchers speak on behalf of participants rather than letting them participate in the presentation as well. While we did not strive to live up to the participatory ideal in this project, we must remain aware of the fact that we have condensed the voices of multiple individuals into a simplified presentation of practices and experiences that are constantly in flux, and that we have made the choice to do so on behalf of the participants rather than jointly with them.

10.7 CONCLUSION

Research on collaborative writing often aims to provide guidelines and solutions that address a particular subset of co-writers’ needs and practices. However, research also continues to unearth difficulties or unmet needs similar to those described two, and even three, decades
ago. Based on our three-stage co-design study we have presented five themes of challenges that reveal a number of contrasts within collaborative academic writing practices and show that, indeed, co-writers do not always agree in their collaborative writing needs and desires. Individual co-writers, even within the same or similar communities of practice, have idiosyncratic and diverse practices and preferences. With their descriptive nature, guidelines fail to capture such contrasting needs and desires. We propose supplementing guidelines with an epistemologically generative framing by focusing on how co-writers’ different needs relate to and influence each other. We have presented a protocol to examine and draw on contrasts when addressing the challenges experienced by co-writers. This trade-off protocol leaves room for others to identify a different set of challenges and contrasts and may prove useful in different collaborative settings; Being characterized by contrasting needs and preferences is likely not unique to collaborative writing but a feature of many collaborative practices. We have started with collaborative writing and encourage researchers and designers to explore this and other collaborative practices through contrasts by involving users in reflecting on individual preferences and trade-offs.

ACKNOWLEDGMENTS

We are grateful to the people who took the time to participate and share their thoughts and ideas with us — this work would not have been possible without you. We thank Susanne Bødker for the suggestion to do this study and for her feedback and guidance. We would also like to thank Anke van Oosterhout for her advice and feedback regarding facilitation of the ideation processes in the workshops and for feedback on the paper, and Henrik Korsgaard for an enriching discussion about the contribution. Thank you also to Mirzel Avdic for feedback on the paper. Finally, we appreciate the suggestions and methodological reflections from the anonymous reviewers.

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 740548).

APPENDIX

The appendix of this paper is included as Appendix A.
LESSONS LEARNED FROM USING REPROGRAMMABLE PROTOTYPES WITH END-USER DEVELOPERS

—

MARCEL BOROWSKI,† Aarhus University, Denmark
IDA LARSEN-LEDET,‡ Aarhus University, Denmark

ABSTRACT

Involving end-users in the development of a product before it is deployed has great potential to increase the fit between a product and individual users’ needs. While end-users can be directly involved in modifying low-fidelity prototypes, they are left out when it comes to high-fidelity interactive prototypes — in part because these cannot be modified directly or require time-consuming edit-compile-run cycles. High-fidelity prototypes, however, are more engaging for users. We created a reprogrammable high-fidelity prototype and explored its use in short-term prototyping workshops with end-user developers, i.e. end-users with programming experience, in the domain of collaborative writing. We report observations and pitfalls, and distill four lessons learned into guidelines on how to use reprogrammable high-fidelity prototypes with end-users in contexts with limited resources. Our experiences demonstrate, among other things, that reprogrammable high-fidelity prototypes are difficult to work with — even for experienced programmers — and emphasize the need for careful attention to guiding participants, time for familiarization, and catering to multiple levels of programming experience.

11.1 INTRODUCTION

Prototyping plays an essential part in the design and development of software. Approaches like, for instance, co-design and participatory design recommend involving users as experts of their own experience [39, 270]. Prototypes can be either low-fidelity (e.g., mock-ups or storyboards) or high-fidelity (e.g., interactive applications). Colombo and Landoni’s study [65] on co-designing with children found that

‡ Both authors contributed equally to this research.
high-fidelity prototypes are more engaging and allow immediate feedback on and evaluation of designs. This immediacy is important as it allows end-users to reflect on the interactive design and provide valuable input while it is still possible to make changes [42]. In contrast to highly flexible paper mock-ups that can be quickly manipulated but only offer limited engagement with the details of interactive behavior, high-fidelity prototypes are resource-intensive and usually need to be prepared by the researchers or designers in-between sessions with users [42, 43], or require edit-compile-run cycles that result in waiting time and can cause participants to become disengaged [42]. This can be especially challenging in explorative research, where resources for implementing prototypes are limited and where there are often harsh constraints on how much time volunteer participants have for co-design and evaluation.

Making software more malleable, i.e. enabling users to modify and reprogram it at runtime, is one step towards mitigating this and allows for prototypes that are simultaneously high-fidelity and highly flexible. While approaches have been taken, in which reprogrammable prototypes are utilized in the design process (e.g., [31, 180, 322]), guidelines on what to consider when using reprogrammable prototypes, in particular in situations with limited resources, are lacking.

As part of a longer project on co-designing for collaborative academic writing [175], we created a prototype that was extensible and reprogrammable at runtime and deployed it in two short-term prototyping workshops. A total of 13 participants worked on collaboratively tailoring and reprogramming the functionality of their prototypes as end-user developers [106]—end-users with some level of technical and/or programming knowledge.

This paper documents successes and breakdowns in our use of reprogrammable prototypes in short-term prototyping workshops in experimental research design processes. We report observations of participants’ work with the prototype during the workshops and provide an overview of breakdowns and lessons learned divided into four themes. By collecting and framing our insights as lessons, we aim to help others get off to a good start when conducting prototyping workshops involving reprogrammable prototypes. Each theme presents recommendations and suggestions for conducting similar prototyping workshops: (1) Ensuring participants’ familiarization with the prototype, (2) avoiding conflation of platform and prototype, (3) aligning expectations with participants and guiding their exploration, and (4) supporting participants with different levels of programming experience. Further, we found that while the short time frame of our workshops was enough to identify the aforementioned themes, it was too short for participants to engage deeply with reprogramming. Still, our results affirm findings that working on high-fidelity prototypes inspires end-users to propose modifications [42], and inform
11.2 RELATED WORK

Our work is situated in the intersection of malleable and reprogrammable software, as well as cooperative prototyping and malleable prototypes. We describe the two domains and how they relate to our project.

**Malleable Software and Tailoring.** Malleable and tailorable software has been explored in a variety of directions in the past: Cabitza and Simone [59] offered an extensive deconstruction of the term *malleability* and classified features of it such as versatility, interoperability, and extensibility. While malleability describes the properties of a system, *tailoring* describes the act of modifying malleable software. Tailoring can happen at different levels (e.g., [225]), each requiring different levels of technical skill—from customizing parameters to reprogramming existing software to writing code for new functionality. MacLean et al. [197] called this “tailoring power” and identified different roles of users that “live” at these different skill levels. Moving from one level of tailoring, e.g., using configuration menus, to the next, e.g., reprogramming the behavior of the system, usually involves a steep learning curve and makes it difficult to transition from one level to the next. MacLean et al. flattened this curve in their malleable system *Buttons* by providing a wider variety of intermediate-level tailoring techniques. They, further, stress the need for creating a “tailoring culture” besides having a tailorable artifact, as only this would shift expectations of users towards changing their software. They did this by employing a designer as a *handyman*, that works with end-users and takes “careful account of the users’ real requirements” [197]. Other researchers have similarly noted that customization is not an individual activity [199], and yet others have explored how particular users can mediate between end-users and designers and developers as so-called *gardeners* [110] or *end-user developers* [106].

**Cooperative Prototyping and Malleable Prototypes.** In their work about cooperative prototyping, Bødker and Grønbæk [42] argue that users need to experience prototypes hands-on in order to envision and shape future use. They emphasize that interactive high-fidelity prototypes—to a greater extent than mock-ups—allow users to experience dynamic properties. With similar reasoning, Bellucci et al. [31] call for “repurposing malleable technologies” for prototyping in their *Extreme Co-Design* approach, and Maceli and Atwood [198] emphasize that as “our technologies have evolved to begin to support the creation of personalized environments,” so our “co-design method-
ologies must evolve as well, to support today’s designers and end users to work together in creating tools for future end-user crafters.” Zhu and Herrmann [322] demonstrate how co-located prototyping in their web-based MikiWiki environment can support collaboration between designers and end-users.

In our work, we focus on end-user developers and on the tailoring dimensions extension (adding and removing pre-made extensions) and reprogramming (modifying extension code). Our notion of reprogramming is similar to “incremental programming” [68] and, like in cooperative prototyping, the prototype is modified during the workshop.

11.3 CASE

11.3.1 Summary of the Preceding Work

The reflections presented in this paper stem from a project on collaborative writing in academia [175]. The reprogrammable prototype discussed here was implemented based on ideation activities with present and former students as well as academic employees from our local university. The needs and preferences identified in collaboration with the workshop participants were characterized by contrasts among individuals and contexts. These contrasting requirements made a reprogrammable prototype appealing, not only with regard to the prototyping situation but also potentially in an end product where it could enable users to tailor their writing tool to particular situations. For example, one contrast was between desiring a high degree of responsivity among writers or prioritizing writers’ ability to immerse themselves in their individual writing tasks without being disrupted.

The design material we drew on when preparing the prototyping workshop was all made by the participants during preceding workshops and consisted of descriptions of issues and themes they experienced in collaborative writing, as well as notes and sketches describing envisioned solutions.

11.3.2 Prototype

The prototype was implemented using the Codestrates [257] platform that in turn builds on Webstrates [162]. Webstrates [162] is an experimental environment that explores the notion of shareable dynamic media. Its implementation synchronizes the DOM (Document Object Model) of web pages across all clients that are visiting a page with a server. The synchronization allows for real-time sharing, editing, and reprogramming of web pages among multiple users using web technologies such as HTML, CSS, and JavaScript. Thus, Webstrates turns web pages into
substrates, i.e. “software entities that act as applications or documents depending upon use” [162]. Codestrates [257] adds an authoring environment on top of Webstrates, allowing both development and use of applications to happen within one environment. The authoring environment is inspired by computational notebooks (e.g., [166]), where regular text and executable blocks of code are placed within the same environment, and allows live computation within the document—at most requiring a page refresh for code changes to take effect.

Our primary reason for using a Webstrates-based system was the inherent support for real-time multi-user collaborative editing. We made use of this to provide participants with a barebones text editor to build on. By using Codestrates, we were furthermore able to make runtime changes easily accessible. This was a highly desirable property as we aimed to create a prototype that was both stable and “flexible enough to allow in-session modification” [42] to mitigate the otherwise known problem of participant disengagement (see Section 11.1).

11.3.2.1 The Prototype Writing Tool

The prototype consisted of a basic collaborative writing tool with the possibility of adding a number of extensions. Although the participants could implement their own extensions, this was not realistic with the time allotted for the workshops. Before the first prototyping workshop, we had therefore implemented a set of extensions based on the preceding ideation activities (cf. Section 11.3.1). In addition to allowing the prototype to be reprogrammed directly in the browser, Codestrates has a built-in package manager [52] which provided facilities for handling extensions.

The basic writing tool only supported simultaneous writing and simple text formatting. Each extension introduced what may be understood as a “feature”—e.g., a button for changing the appearance of a paragraph or a tool that would suggest phrases (see Table 11.1). The prototype was self-contained and each group worked on their own copy of it, which could be immediately used and modified. Reprogramming was possible at the level of appearance (HTML and CSS) and behavior (JavaScript). While extensions could be reprogrammed just like the barebones writing tool, they were additionally intended to allow participants to easily add and remove features in order to build up a custom writing tool in a plug-and-play fashion that did not require programming. We had implemented a total of ten extensions that were all available to the participants (see Table 11.1).

11.3.2.2 Modifying the Prototype Substrate

In addition to each substrate being reprogrammable, it is also possible to extend Codestrates’ own functionality from within itself (see [257]). This blurs the boundary between Codestrates, as a platform on which
**INSPIRATION PROMPT**
Provides suggestions for phrases in order to combat writer’s block.

**META NOTES**
Attaches a note to a whole paragraph. Three categories of such notes are available: overview, general notes, and revisions.

**PARAGRAPH VOTING**
Lets writers approve paragraphs as “finished” by voting. A paragraph with an approval vote from all writers is framed by a golden border.

**PARAGRAPH STATE**
A button that toggles through different appearances of a paragraph to express its state: not set, draft (typewriter font), review (serif font), or finished (serif font and golden frame).

**WHERE AM I NEEDED?**
Shows a list of paragraphs with few revisions or no recent revisions. Clicking on an entry takes the writer to the given paragraph.

**REVISION OVERVIEW**
Overview of the revision history in a stylized miniature of the document, that can be color-coded according to four criteria: number of edits, last edit time, first edit time, or whether or not a particular user has edited the paragraph.

**REVISION OVERLAY**
Visualizes revision information as lines of varying thickness and color next to each paragraph, showing either: whether a particular user has edited the paragraph or not; relative total edit count; or relative time of editing.

**REVISION DISPLAY**
A display next to each paragraph, showing the number of edits made to that paragraph. Clicking the display shows number of edits per user, with timestamps of their first and last edits.

**PARAGRAPH LOCKING**
Lets writers block co-writers from editing a paragraph and partly or fully obscure the contents of it.

Table 11.1: Extensions available for participants to add to their prototypes.

---

the prototype is implemented, and the prototype as a substrate that relies on the Codestrates platform. For this study, we understand *Codestrates* as the architecture providing extensibility through its package management and reprogramming capabilities. By *prototype* we refer to the writing tool, including any modifications or extensions added to augment it.

Since Codestrates follows the structure of computational notebooks, it contains input areas for regular text as well as for blocks of program code. We utilized this in the prototype by letting the Codestrates text editing fields make up the text area of the writing tool. This meant that the text editing facilities that were part of the prototype’s design were placed adjacent to (and potentially intermingled with) the implementation code for the prototype and any included extensions.
11.3 CASE

Figure 11.1: Screenshots of the prototype’s two views. The views can be switched with the “Switch Theme” menu entry on the left.

Both the content in the regular text areas and the implementation code could be edited collaboratively in real-time. To aid participants in distinguishing the activity of implementing functionality from that of trying out the prototype, we had implemented two views: One for programming, which included features such as disabling packages or running code, and one intended for writing in which the only UI elements visible were the ones that were part of the prototype’s user interface, e.g., text formatting (see Figure 11.1 and the accompanying video).

11.3.3 Workshop Procedure

These prototyping workshops were intended (1) for participants to see their design ideas come to life in a high-fidelity prototype that still permitted further ideation and refinement during the workshop, and (2) for us to identify potentials and breakdowns of using reprogrammable prototypes in short-term workshops. The participants were divided across two identically structured workshops in order for us to accommodate a large number of participants (N = 13) and still be able to interact with each group and help them out when needed. Both workshops lasted two hours and were facilitated by the authors. The short time frame was due to limited project resources, which meant that participants were uncompensated volunteers. While the time frame was only enough for a small amount of reprogramming, it was enough for us to identify a number of lessons that others with similarly limited resources can learn from.

The workshops were both structured as follows: First, the Codestrates platform and its package management were briefly explained to participants. Next, we presented the prototype and demonstrated
how to add and modify extensions. We, further, showcased all ten extensions and outlined their functionality. The whole introduction lasted 20 minutes and took place in plenary using a large display to do a live demo. To supplement the introduction, and for reference throughout the workshop, participants were each given a pamphlet containing a description of each extension as well as instructions on how to access, use, and modify the prototype. After the introduction, we divided the participants into groups: three groups per workshop, six groups in total (see Table 11.2). The participants used their personal laptops to access and work on their group’s copy of the prototype.

First, groups were instructed to explore the prototype and try out the extensions they found most interesting. Following a few minutes of exploration, the groups were instructed to each select one of three writing scenarios: (a) jointly writing a master’s thesis, (b) remotely authoring a book, and (c) co-authoring a scientific paper. They were tasked with tailoring the prototype to fit their scenario. The scenarios were meant to guide participants’ work with the prototype by providing them with a focus. While using and modifying the prototype, each group also created storyboards on paper to document ideas for features and modifications. This was intended to avoid ideation being hindered by technical obstacles, such as not knowing how to implement an idea. Each group was also given and encouraged to continuously fill out, a free-text questionnaire about the prototyping workshop and the project as a whole (see also Section 11.3.5).

The prototype exploration and modification activity lasted one hour. 30 minutes into it, we introduced a set of changes to the scenarios. These disruptions were meant to push participants to reconsider assumptions at play in the modifications they were making. After working with the prototype for one hour, the groups took turns presenting what they had worked on. This was followed by a plenary discussion about reflections spurred by the questionnaire.

11.3.4 Participants

13 people participated in the workshops. We recruited participants from our local university. The only requirement for participation was current or recent experience with collaborative academic writing. Experience with programming was not a requirement. Participants were between 18 and 55 years old, three described their gender as female and ten as male. All but one participant were studying or working in a technology- and/or design-focused field.

Each participant filled in an anonymized demographic questionnaire when they entered the project. This information was used for providing the overview of participants’ backgrounds in Table 11.2. The table also contains a categorization of how often the participants did programming in their daily life at the time of the prototyping
Table 11.2: Overview of the the groups participating in the prototyping workshops. G1–G3 participated in the first workshop, G4–G6 in the second.

workshop (similar to [246]). This is based on short meetings we had with each participant prior to the prototyping workshop in order to group them based on programming experience (both frequency and kind of experience, e.g., web development vs. hardware). This was done with the expectation that grouping participants with similar skill sets would ensure that all group members felt like they were able to contribute. The categorization has been confirmed with participants for the reporting in this paper, resulting in a few minor adjustments; however, none that would have impacted the groupings.

11.3.5 Data Collection and Analysis

We collected the following artifacts that participants worked on during the workshops: storyboards, their modified copies of the prototype, the mock documents used during prototyping, i.e. the text being edited when trying out the prototype, and the free-text questionnaire. The data obtained from the free-text questionnaire comprised answers to ten questions, of which the ones relevant to the present paper asked about the prototyping process: In particular, whether participants experienced any confusion or surprises and if they could think of any customization options that were missing. We additionally took photos throughout the workshops, and video and audio recorded the plenary activities and the dialogues of each group (using one stationary camera per group). The recommendations in this article are based on observations we made during the workshop, supplemented with the data mentioned above. The photos have primarily been used for dissemination about the workshops.
We conducted a qualitative analysis with the recordings as our primary source. Using the video material, incidents in the group work were categorized according to the type of episode (with the five labels: discussion, coordination, prototype work, problem, something unclear) through meaning condensation [172]. We supplemented this by examining the groups’ modified prototypes and storyboards. The plenary discussions were transcribed as part of the overall project, presented elsewhere [175]. Our analysis was further informed by the questionnaire and the transcripts of the plenary discussions.

11.4 Observations

In this section, we describe for each group, first: what modifications participants made to their prototype instances, and second: what their process was like during the workshop. We refer to participants as P1–P13 and to groups as G1–G6. During the two workshops, five out of the six groups reprogrammed the prototype. Three of these groups did so by changing functionality using JavaScript (G2, G3, G6), one group focused on modifying appearance using CSS (G5), and one group attempted to modify the prototype but did not succeed (G1).

G1 (thesis scenario). G1 attempted to create an extension for planning the content of paragraphs, a “planning state” (G1). Their idea circled around initial phases of writing, where this extension would prompt authors to plan each paragraph’s content using a list of bullet points.

G1 began by looking through the pamphlet and discussing which extensions they liked, before installing a couple (Paragraph Locking and Meta Notes) and playing around with them while discussing ideas based on them. Most of the exploration was done separately, with occasional verbal exchanges and use of the pamphlet. They spent quite some time trying to figure out how to get started with programming, mainly talking about how to do it rather than actually doing any programming. It took a while for them to figure out how to access the code for the different extensions, and they generally struggled with a lack of overview of the implementation, having trouble locating things in the code or being surprised that making changes to the rich-text implementation would affect all running instances of it. Another problem seemed to be that Codestrates violated their expectations from previous programming experience, such as when P1 made a remark about not knowing how to do something “in onCreate,” i.e. when the prototype is loaded. A couple of times, they remarked on not having enough time to do what they wanted, and they ended up deciding to mock up their desired feature instead of fully implementing it. In the plenary session, they explained that they had missed the Paragraph
States extension as an obvious starting point “due to the fact that there are so many packages” (referring to the extensions).

**G2 (thesis scenario).** The participants in G2 mostly worked individually. While P4 (unsuccessfully) attempted to modify the Revision Display to show the number of edits contributed by each individual author, P3 modified the “Where am I needed?” extension to make it possible to tag co-authors in a paragraph: Authors could then be tagged using the notation @username within a paragraph, upon which the paragraph would be listed in the “Where am I needed?” dialog box (see Figure 11.2a).

G2 began their work by individually reading the pamphlet. Early on, they clarified with a facilitator what was “allowed”—getting confirmation that “ridiculous” ideas were also allowed. G2 quickly turned their focus to the code, individually trying to understand it. P4 seemed to struggle with this, making use of the browser’s developer tools and console to explore it: “There is a lot to figure out; where to start and what to do.” After some time of trying out an extension, they drew a storyboard about an idea for a tagging feature. While P3 started working on that feature, P4 continued to explore the prototype and try to understand the code. G2 began modifying code within 20 minutes, unlike the other groups who all spent around half of the allotted hour before starting to do any programming. P3 switched to using the external code editor Atom and copied code back and forth from Codestrates because he found the Codestrates editor to lack essential features, such as advanced find and replace. Finally, the naming of extensions caused some misunderstandings, as P4, e.g., mistook the background service for tracking changes (Revision Tracker) to be the Revision Display extension. As understanding the code in the given time was too complex, P4 ended up creating a storyboard about his idea instead.

**G3 (book scenario).** G3 modified the Revision Overlay extension to not only indicate when the last edit of a paragraph happened, but also by which co-author.

G3 began by trying to become familiar with the system and looking through the pamphlet. G3’s ideas for modifications took outset in the extensions, such as when they discussed how the “Where am I needed?” extension could be modified to allow writers to indicate that they need help from co-writers. When asking a facilitator whether their intended modification would be feasible within the time frame of the workshop revealed their ambitions to be unrealistic, they instead described their idea in a storyboard. While trying to reprogram an extension, G3 struggled to understand the underlying architecture of the prototype—the lack of an inspection feature for the system state was another hurdle. Help from the facilitator was crucial for the group.
(a) G2’s modification, a tagging feature that was integrated into the “Where am I needed?” extension. 

(b) G5’s modification, an updated style of the Meta Notes extension. The blur was removed and the notes box moved.

Figure 11.2: Screenshots of examples of the groups’ modifications. Text content and user names in the screenshots were modified to ensure anonymity.

to succeed in creating their modification. Although programming collaboratively, each group member primarily used their own laptop and occasionally looked at the screen of the other members to see what they were doing. P7 commented on this collaborative coding as being “a completely new experience” and that “it’s kind of weird to be typing around in the same code at the same time.”

G4 (book scenario). G4 did not attempt to modify the prototype beyond adding existing extensions.

G4 started by exploring the prototype, including switching between the writing and programming themes. In doing so, both participants faced minor technical problems such as toggling full screen mode and not knowing how to exit it. P8 also struggled with her laptop running slow, resulting in the prototype becoming unresponsive and requiring P8 to reload the page multiple times during the second half of the workshop. The facilitators lent P8 one of their laptops for the last third of the workshop. Besides experimenting with the provided extensions, the group also inquired about the potential to create images in a document. A facilitator provided this by helping them install a drawing extension from another project. Consisting of two participants that rarely program, G4 focused on exploring existing extensions and making storyboards.

G5 (thesis scenario). G5 modified the Meta Notes extension. Initially, the extension displayed notes on top of the text and blurred the text underneath. G5 changed the styling of the extension, so that the notes were placed besides the text and the text was not blurred anymore (see Figure 11.2b).
G5 started by skimming the pamphlet to get an overview of the available extensions. Initially, they had some problems with basic functions of the prototype, such as creating new code blocks. They also required help from a facilitator to find the CSS code to modify. After making their modification, G5 discussed possible ways of using other extensions in their scenario. Because of the limited time, they sketched their ideas on paper and using the drawing extension within the prototype (inspired by G4). In the second half, one participant experienced a bug that caused the prototype to freeze, but this was resolved by a facilitator.

**G6 (thesis scenario).** G6 updated the Paragraph State extension to display what state a paragraph is in (draft, review, or finished) with a text label instead of only indicating it through styling.

G6 began by discussing which extensions to install, looking through the package manager and the pamphlet. From early on, they anchored this discussion in their chosen scenario. They decided to work towards their idea by modifying Paragraph States. This spurred further elaboration, in the form of improvements they wanted to make to Paragraph States. When it came to exploring implementation details, P13 was quick to get a grasp of both the Codestrates user interface (such as how to access the implementation code) and its inner workings, and on a number of occasions would explain things to P12 when he expressed confusion. They were both slowed down by being restricted to JavaScript and CSS, since they were familiar with other languages and programming libraries that they would have liked to draw on. They interacted with a facilitator on multiple occasions, to discuss how to accomplish particular things. Along the way, they sometimes used a storyboard to communicate about their ideas. Making a storyboard later became a way for P12 to contribute when P13 wanted to focus on programming while P12 was not sure what to do. For the majority of the time, they worked separately on each their laptop. The real-time shared editing caused some trouble, with P13 asking P12 to refrain from writing anything until P13 was done making some edits to the code “because my [caret] is flying around when you do it.”

### 11.5 Lessons Learned

We now report on four lessons learned from using a reprogrammable prototype during short-term prototyping workshops. Based on our experiences, we provide suggestions and recommendations in the boxes at the end of each subsection to guide the design of studies and prototypes. As mentioned, while the time frame we had available was not enough for substantial reprogramming, it was sufficiently long for us to identify a number of breakdowns. These recommendations will not—and are not intended to—overcome this issue or the lack
of a tailoring culture [197]. Rather, they aim to inform about what breakdowns can occur and how to mitigate them in similar workshops.

11.5.1 **Familiarization**

Many of the problems that participants faced originated in confusion about the underlying Codestrates platform. G6, for instance, had trouble finding the implementation code they wanted to change, and G4 initially struggled with adding extensions. The real-time collaborative editing, further, meant that modifications were instantly shared within the groups, which presented participants with both advantages and challenges. For example, many groups would switch quite fluently between focusing on individual activities and asking questions regarding things other group members were currently doing. But we also saw several instances in which the real-time sharing caused visual disruption or broken code.

After some time familiarizing themselves with the prototype, the participants could largely overcome these issues. However, the fact that most groups spent so much of the time allotted for modifications on getting familiar with the prototyping environment indicates that we underestimated the necessary scope of familiarization. Although participants did explore the extensions while getting familiar with the environment, the level of extension resulting from this was minimal due to the overhead of figuring out how to add and use extensions, and only G2 spent the majority of the time reprogramming. Letting participants interact with the prototype during the demo (see Section 11.3), could likely have revealed confusion and helped facilitators resolve it earlier in the process. Providing a pamphlet as written documentation did support familiarization and allowed participants to read up on things later but could not stand on its own.

- Ensure that participants understand central principles of the prototyping platform, e.g., through hands-on tasks.
- Engage participants actively when introducing the prototype, e.g., through a tutorial.
- Provide simple written and illustrated documentation for participants.

11.5.2 **Conflation of the Platform and the Prototype**

Switching back and forth between development and use led participants to focus on Codestrates and technical issues rather than on the prototype—an issue also found in earlier studies where “discussions quite easily get focused on the current prototype and rather technical
issues” [42]. As described in Section 11.3.2, reprogramming took place directly in the prototype’s user interface. While this is beneficial in that there is no need for external applications like an IDE, this paradigm is rather different from conventional development paradigms and is hence unfamiliar even to many experienced programmers. We suspect this was the reason for participants’ conflated understanding of the underlying Codestrates platform and the actual prototype. This conflation also surfaced in the plenary discussions, where participants would not distinguish between their experience of using Codestrates and discussing ideas for prototype designs. There is a trade-off in using a novel development paradigm that allows new and possibly better ways of reprogramming, but which may also prevent participants from reusing previous experience and cause the platform to become the focus of discussion instead of the prototype design. If quick engagement with reprogramming is desirable, using familiar development paradigms should be considered.

- Consider whether the development environment should be separate from the prototype.
- If relevant, explain the interwovenness of using and developing the prototype in the same environment.
- Demonstrate the borders between the platform and the prototype and facilitate exploration of them before participants start developing.

11.5.3 Aligning Expectations and Facilitating Exploration

We experienced a number of situations in which we had not managed to adequately guide participants’ expectations for what would be achieved in the workshop. For one, most groups struggled with implementing desired changes within the allotted time frame, among other things because they jumped straight to plans for major changes instead of starting by exploring minor adjustments. Some participants downplayed what they had accomplished, such as when P3 presented what we saw as a successful modification as “a half-assed implementation of something.” In addition to frustrations caused by the short time frame, we suspect that participants’ expectations for themselves are connected with assumptions about the facilitators’ expectations and how to live up to those. P4 asking what was “allowed” exemplifies how participants’ work is guided by trying to fulfill facilitators’ expectations. We do not see it as problematic that participants wanted to align their goals with our expectations but include it as an example of the importance of clear communication about expectations. The scenarios seemed to provide guidance for some participants, but
our experiences show that they may need supplementing with more explicit communication about expectations.

- Make clear to participants what kinds of modifications are possible, what will be feasible in the allotted time, and what would be a satisfactory outcome. Note that the purpose is not to set a standard but to make participants understand that small contributions are fine.

- Walk participants through different levels of what is possible, e.g., progressing from changing styling to changing the behavior of an extension.

- Provide concrete tasks or scenarios to solve and/or discuss.

- Provide clearly available options for exploring the prototype, e.g., through configuration options or highlighted parts of the code.

11.5.4 Catering to Different Levels of Programming Experience

How frequently participants programmed in their day-to-day life varied from programming daily, e.g., as a software engineer, to only programming rarely. From the provided extensibility through the package management and reprogrammability of code, all groups used the package management while exploring different features, but use of reprogrammability was dependent on the participants’ programming skills. All groups except G4—the only group without a participant who programs regularly or daily—engaged in reprogramming extensions. While G5 focused on style changes, the remaining groups changed the functionality of extensions. The provided extensions supported participants in coming up with ideas and supported dialogue—e.g., G3 discussed improving the Revision Overlay extension: “So what can we do with this overlay, how can we make it more interesting?” (P5). Our observations additionally confirm that providing pre-implemented extensions was necessary, given the limited time frame.

We see both opportunities and difficulties with our approach: For participants with programming experience, reprogrammability can be motivating and spark creativity. For participants without much programming experience, however, the learning curve was too steep and did not allow them to engage in modifying extensions—this echoes the results of [197]. Those participants need additional tools that support reprogrammability in a more accessible way, for instance by having a GUI for configuration of, e.g., styling and parameters, or by enabling direct manipulation for configuring the prototype [42].
11.6 Limitations and Future Work

Such directly available options could also be useful for spurring exploration (cf. Section 11.5.3).

- Provide different levels of tailorability suitable for the participants, e.g., by supplementing scripting with a GUI for configuration.
- Create a starting point, such as extensions that participants can build on instead of starting from scratch.

11.6 Limitations and Future Work

This project had a number of limitations. First, only 3 of the 13 participants were women. Research indicates that gender can impact the way people approach end-user programming environments (e.g., [30]). With a different gender distribution, and perhaps all-women groups, we may have uncovered more or different issues and seen more diverse examples of resourcefulness from participants. Second, the short duration of the workshops meant limited time for both familiarization and modification. Some of the presented issues can likely be alleviated with longer workshops and/or several workshops with the same participants. Third, (real-time) collaborative writing as a use case is relatively close to real-time collaborative programming, which likely added to the conflation of prototype and platform observed in participants’ discussions. Our suggestions for guided familiarization may reduce this, but it may also be that the mode of collaboration needs to be considered more carefully against the use case. Finally, preparing the prototype in advance of the workshop involved a trade-off: While it reduced downtime for debugging or larger implementation changes, in which participants could become disengaged, it also excluded participants from an implementation step that could have given them a greater sense of agency and ownership in the process. In addition, we note that our guidelines are aimed at research on the use of reprogramming. Product-oriented user-involvement likely requires other steps.

11.7 Conclusion

This paper illustrates breakdowns and pitfalls of using reprogrammable prototypes in prototyping workshops with limited resources and offers guidelines on how to mitigate them. Based on two prototyping workshops, we have compiled suggestions under four themes: (1) Ensuring participants’ familiarization with the prototype, (2) avoiding conflation of platform and prototype, (3) aligning expectations with participants and guiding their exploration, and (4) supporting partic-
participants with different levels of programming experience. We believe some of the suggestions are also applicable for other user groups than end-user developers and encourage more research to explore such workshops with different user groups. We also generally encourage the employment of reprogrammable software in prototyping processes to engage end-users more deeply with the interactive behavior of prototypes, while keeping in mind the limitations of short-term workshops like ours.

ACKNOWLEDGMENTS

We are grateful to the people who took the time to participate and share their thoughts and ideas with us, both during and after the workshops — this work would not have been possible without them. We also thank our colleagues for feedback and insightful discussions: Peter Lyle, Philip Tchernavskij, Carla Griggio, Tiare Feuchtner, and Wenkai Han, as well as Susanne Bødker and Clemens N. Klokmose, who have provided guidance throughout the writing of this paper. Finally, we appreciate the feedback and suggestions from the anonymous reviewers.

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 740548).
CODESTRATES V2: A DEVELOPMENT PLATFORM FOR WEBSTRATES*

—

MARCEL BOROWSKI, Aarhus University, Denmark

JANUS BAGER KRISTENSEN, Aarhus University, Denmark

ROLF BAGGE, Aarhus University, Denmark

CLEMENS N. KLOKMOSE, Aarhus University, Denmark

ABSTRACT

This report presents the Codestrates v2 development platform for Webstrates. Webstrates is an experimental platform to explore the vision of computational media. Codestrates v2 consists of three components: The Webstrates Package Manager, the Execution engine, and the authoring environment Cauldron. Together, the three components create a powerful development platform for Webstrates on top of Webstrates. This report summarizes the background of the project and its original motivation. We introduce the three components, explain their functionality and use, and provide details about how they are implemented. Finally, two example cases illustrate how the platform allows to create a simple todo list application and a computational notebook.

12.1 INTRODUCTION

This report presents Codestrates v2, a development platform for creating collaborative web applications with Webstrates [162]. Webstrates is an implementation of the idea of computational media on the Web: software that is distributable across devices, shareable with other users, and malleable to let users modify their software and tools to their personal preferences. Codestrates v2 generalizes principles introduced in Codestrates v1 as a general purpose framework for building software on top of Webstrates. Codestrates v1 [257] is a research prototype that provides an authoring environment for Webstrates in the form of a collaborative computational notebook environment. It follows the

The literate computing paradigm [248] and structures code into paragraphs that can be organized in sections. Codestrates v1 featured a notebook-based editing environment, a model for polyglot code execution, e.g., using both JavaScript and Python code in the same notebook, and a package manager [52] that enables users to easily share content and tools across different notebooks. The components introduced in this report build on top of Webstrates and aim to replace Codestrates v1 in a more generalized and robust fashion. Codestrates v2 consists of an Execution Engine, the Webstrates Package Manager and the Cauldron authoring environment (see Figure 12.1). Together, the three components allow users to collaboratively create web applications in Webstrates directly inside the web browser.

Webstrates Package Manager. The Webstrates Package Manager (WPM) enables easy sharing of code between different documents. WPM takes the ideas of packages as found in Codestrates v1 [52] and generalizes the idea towards sharing arbitrary code and assets as packages between webstrates. This decouples the WPM component and makes it independent of the rest of Codestrates v2.

Execution Engine. The execution engine component, next, manages code execution and code editing. It generalizes the ideas of Codestrates v1 [257] by introducing the notion of code fragments: pieces of code that can be executed and modified individually. In contrast to Codestrates v1, which combined code execution framework and authoring environment in one component, the engine of Codestrates v2 is independent of an authoring environment and instead provides an API for instantiating code editors in a user interface that can be used in an authoring environment.

Cauldron. Cauldron, lastly, is the first authoring environment we created for Codestrates v2. It allows users to author their applications right within the web browser. It provides a familiar IDE-like experience with a tree browser and tabbed editors.

The remainder of this report is structured as follows: Section 12.2 introduces the Webstrates and Codestrates project and how they influenced the development of Codestrates v2 and Cauldron. Section 12.3, Section 12.4, and Section 12.5 summarize the Webstrates Package Manager, the Execution Engine, and the features of the Cauldron authoring environment. Section 12.6 illustrates examples of how Codestrates v2 and Cauldron can be used to create applications. Section 12.7 details how the platform is implemented, and Section 12.8 concludes the report and hints towards possible future work.
12.2 BACKGROUND

This report continues the effort of the Webstrates [162] and Codestrates [257] projects to create experimental software platforms to investigate computational media [233]. The work on Webstrates is inspired by an attempt to create a modern take on Kay and Goldberg’s idea of personal dynamic media [152]. Personal dynamic media describes a software medium that is malleable and can be modified by end-users — Kay and Goldberg compare it with mediums such as “paper and clay.” We introduced the idea of shareable dynamic media [162] by articulating four key properties: shareability, being shareable between users and allowing for real-time collaboration, distributability, allowing documents to be distributed across heterogeneous devices, and computability, allowing documents to execute arbitrary code. Recently, we have been using the concept of computational media for reasons of simplicity and will continue to do so in this report. One of the characteristics of computational media is that it dissolves the traditional boundaries of what is an application and what is a document. A computational notebook is an example of computational media.

12.2.1 Webstrates

Webstrates [162] is a prototype platform for computational media. Webstrates is built on standard web technologies and functions as a
web server. A document in Webstrates—a webstrate—can be opened in a regular web browser. A webstrate is identified by its URL. In contrast to a regular web page the DOM (Document Object Model) of a webstrate is synchronized in real-time with the Webstrates server and between all clients that access a webstrate. This synchronization includes HTML, CSS, and JavaScript that are part of a webstrate. Users can modify the JavaScript in the DOM in one client, e.g., through the developer tools of the web browser, and changes are synchronized immediately in all other clients, allowing to create collaborative applications in the web.

Using transclusion, webstrates can be embedded into other webstrates and composed flexibly. Transclusion in Webstrates uses the iframe (inline frame) HTML element. A transcluded (child) webstrate can communicate with the transcluding (parent) webstrate and exchange information.

Webstrates, furthermore, supports a variety of APIs\(^1\) including assets storage, signaling between clients, or transient DOM elements or attributes that are not synchronized with the Webstrates server. Transient elements can be used to create user interfaces that should not be synchronized across multiple clients—enabling more relaxed WYSIWIS (What You See Is What I See) user interfaces.

12.2.2 Codestrates v1

Codestrates v1 [257] is an authoring environment built on top of Webstrates. Codestrates v1 implements a GUI inside the web browser that allows to author webstrates and create applications directly in the browser (see Figure 12.2). A document in Codestrates v1 is called a codestrate. The authoring experience of Codestrates v1 was inspired by computational notebooks and the literate computing paradigm [248], which describes the interweaving of code with descriptive text within the same environment. While computational notebooks were usually used for data analysis and visualization purposes, Codestrates v1 allows to organize applications in “paragraphs,” e.g., JavaScript, CSS, HTML, and JSON paragraphs. Paragraphs, again, can be structured in sections (see Listing 12.1).

To support sharing of content or functionality across multiple codestrates, we introduced a package management [52], which was inspired by package managers in, e.g., Node.js or Visual Studio Code. Packages in Codestrates v1 are sections with a particular attribute (see Listing 12.1). When installing packages, the package management transcludes another codestrate, copies over relevant packages, and then removes the transcluded codestrate again.

---

\(^1\) Webstrates APIs: https://webstrates.github.io/userguide/api.html
(Retrieved February 22, 2022)
12.2 Background

![Screenshot of Codestrates v1](image)

Figure 12.2: A screenshot of Codestrates v1. The user interface is inspired by computational notebooks and structured in sections and paragraphs (from [257]).

12.2.3 Limitations of Codestrates v1

Codestrates v1 was used in multiple research projects including collaborative data visualizations [16], collaborative video editing [164], collaborative programming assignments [53], and a “computational labbook” for nanoscience [233]. Throughout these projects, the execution model of Codestrates v1 and its ability to author and use software within the same environments enabled users to use and modify applications without the need for additional software. The computational notebook user interface and the structure of content into paragraphs and sections, however, was limiting the expressivity within Codestrates as well as its loading time: Being forced into a one-dimensional linear notebook environment was a limitation both when authoring software, as it, for example, was not possible to open two paragraphs of code side-by-side, and when using software, as the user interface of Codestrates v1 was always loaded and thereby interfering with styling of applications and their loading time.

Codestrates v1 was implemented in itself, meaning that the implementation of new features happened inside the notebook itself. While this nicely illustrates the malleability of the medium, for Codestrates v2 we hired professional programmers for the development and they needed to be able to use their conventional tool chain (NetBeans IDE with refactoring tools, version control, etc.) to be productive. Hence, software built with Codestrates v2 is reprogrammable from within the webstrate, however, the three core components (Webstrates
Listing 12.1: The DOM structure of a codestrate in Codestrates v1. All paragraphs need to be placed within sections and sections within the element with ID sections. A package in Codestrates v1 consists always of a single section, e.g., line 5–9.

Package Manager, Execution Engine, and Cauldron) require external development tools to be modified.

12.3 Webstrates Package Manager

The Webstrates Package Manager (WPM) is inspired by the Codestrates v1 package management system [52]. WPM allows to import and reuse code from other webstrates. Using WPM, it is possible to define a bundle of DOM elements and assets to a package and provide it with metadata including description, version number, author, and dependencies to other packages.

In contrast to the package management of Codestrates v1, WPM does not include a user interface. To use WPM, JavaScript functions have to be called directly, for instance, in scripts or using the developer console (see Listing 12.3).

12.3.1 WPM Packages

WPM packages exist in the DOM of a webstrate and are defined using custom `<wpm-package>` element. All child elements inside `<wpm-package>` are part of the package and the `id` attribute is used to uniquely identify a package (see Listing 12.2). Inside another custom element, the `<wpm-descriptor>`, it is possible to define meta data of a package. The descriptor contains a serialized JSON object with the following properties:

- **description**: A description of the package.
• dependencies: Selectors for other packages that this package depends on.

• assets: Assets that the package uses. These will be copied over with the package when it is installed in another webstrate.

• version: The current version of the package.

```
<wpm-package id="myPackage">
  <wpm-descriptor>
    {
      "description": "My Package",
      "dependencies": [ "#myOtherPackage" ],
      "assets": [ "myImage.png" ],
      "version": "1"
    }
  </wpm-descriptor>
  <script>
    console.log("Hello from myPackage!");
  </script>
</wpm-package>
```

Listing 12.2: A simple example WPM package.

12.3.2 Requiring Packages

Installing packages in WPM is done by requiring them using the `require()` function: One webstrate serves the package, while other webstrates can require the package from it.

A package can be installed either transiently or persistently (see Listing 12.3). By default packages are installed transiently: the packages are copied into a transient element in the `<head>` element. In this case, packages are retrieved from the repository every time a webstrate is loaded and are therefore automatically updated once the version in the repository is changed. If a package is installed persistently, the package is copied into a given DOM target node and persisted by the Webstrates server. This can be useful, if a specific version of a package is used or users want to modify a package locally in one webstrate, e.g., to test new features. WPM will build a dependency tree so that dependencies are always installed before the dependent package.

12.4 Execution Engine

The development of the execution engine was motivated by the shortcomings of Codestrates v1 described in Section 12.2.3 and set out to decouple the user interface and authoring environment from the execution framework. In this section, we describe the
The execution engine is packaged with WPM and deployed on webstrates. By separating the authoring environment from the execution engine it is possible to author the same code using different authoring environments. While developing Cauldron we, e.g., created a proof-of-concept version of a “linear editor” that displays code in cells like a computational notebook.

12.4.1 Code Fragments

As the naming scheme “paragraphs” in Codestrates v1 was related to its notebook-like interface, we renamed the “computational units” in a document from paragraphs to fragments. A fragment is a DOM element with the tag name code-fragment that has a data-type attribute (see Listing 12.4). The data type describes the type of content of a fragment, such as HTML, Markdown, LaTeX, CSS, SCSS, JavaScript, TypeScript, Python, Ruby, and Lua. Fragments are hidden by default and not rendered in the browser. Content inside a fragment is always stored as plain text in the DOM and is, thus, not interpreted on its own by the browser.

Fragments can be placed anywhere in the DOM and structured arbitrarily, introducing the possibility to create deeper hierarchies (e.g., using <div> elements as folder). Besides this structural flexibility, it further allows to create different editors for editing the content of these fragments: It is now possible to view fragments both in a one-dimensional notebook representation as before, but also as a tabbed editor, similar to an IDE, or a two-dimensional canvas where fragments can be freely moved around.

12.4.2 Executing Code

The computational engine of Codestrates v2 manages the execution of fragments, the creation of virtual machines for executing code, as well as providing functionality to reuse code from fragments using require()—similar to modules in Node.js. Fragments with program code (JavaScript, TypeScript, Python, Ruby, Lua) can be “run” individually by the user to execute their code. If a fragment should be
executed whenever the webstrate is loaded, it can be set to auto-run by adding the auto attribute to the fragment element (see Listing 12.4). A JSON fragment allows to store data inside a fragment (see the example in Section 12.6.1).

JavaScript fragments can run directly in the browser, while other programming languages will require a transpiling step. Thus, Codestrates v2 is polyglot, i.e. it allows fragments to be written in multiple programming languages. Even more, code from one fragment can be exported and imported in another.

```
<html>
  <head> ... </head>
  <body>
    <code-fragment data-type="text/javascript" auto transient-fragment-uuid="134a...">
      console.log("Hello World!");
    </code-fragment>
    <code-fragment data-type="text/html" auto transient-fragment-uuid="103c...">
      &lt;p&gt;Hello World!&lt;/p&gt;
    </code-fragment>
    <transient id="103c..." class="autoDom">
      <p>Hello World!</p>
    </transient>
  </body>
</html>
```

Listing 12.4: Example of a JavaScript and HTML code fragment. Both fragments are set to auto-run, indicated by the auto attribute. The HTML fragment renders its content into a transient element. The code fragment is linked to the rendered element using a transient ID that is created on page load.

12.4.3 Enabling Markup and Styling

Fragments with markup (HTML, Markdown, LaTeX) and fragments with stylesheets (CSS, SCSS) can only be set to auto-run and not run manually. Content from markup and stylesheet fragments is rendered or preprocessed into a transient element with the class autoDom, which is linked to its respective code fragment using an ID (see the <transient> element in Listing 12.4). Content is synchronized from the fragment to the transient rendering so changes to the content of the fragment are automatically applied in the rendering (for markup and stylesheet fragments). By rendering HTML in a transient element opposed to changing it directly, it is possible to collaboratively edit the fragment code. This in turn, however, also means that changes to the rendered DOM elements are not synchronized back to the fragments, as this could cause conflicts (see Figure 12.3).
12.4.4 Creating Editors

The execution engine also provides functionality to create *code editors* and view-only editors, or just *views*, for fragments that are synchronized with the fragment content. Listing 12.5 shows an example of how a code editor can be created for a fragment. The engine currently supports three types of editors that build on their respective libraries: Ace, CodeMirror, and Monaco.

Besides code editors, the engine can also create *views*, which render markup fragments (HTML, Markdown, LaTeX) into an a read-only editor instance. The view is updated live whenever the fragment changes. The view is, for example, used in Cauldron to view a live preview of a fragment or in the Ganymede example (see Section 12.6.2) to display rendered Markdown fragments. If a markup fragment is set to auto-run a view will automatically and transiently be created as an immediate sibling to the fragment in the DOM (see Listing 12.4).

The creation of editors is part of the fragment and editor engine as it provides a basic way of editing fragments that handles the synchronization between the fragment and the editor. Authoring environments—like in our case Cauldron—can then use these editors and display them.

12.5 CAULDRON

Cauldron is an authoring environment for Codestrates v2 that allows, e.g., to manage and edit fragments. In contrast to Codestrates v1, Cauldron does not follow a computational notebook interface but the interface of a normal tabbed editor, where tabs can be arranged

---

2 Ace Editor: https://ace.c9.io/ (Retrieved February 22, 2022)
3 CodeMirror Editor: https://codemirror.net/ (Retrieved February 22, 2022)
4 Monaco Editor: https://github.com/Microsoft/monaco-editor (Retrieved February 22, 2022)
```javascript
let fragment = Fragment.one("#myFragment");
let editor = EditorManager.createEditor(fragment, {
    editor: MonacoEditor,
    theme: "light",
    mode: "full"
});
let editorNode = editor[0].html[0];
document.body.append(editorNode);
```

Listing 12.5: Creating an editor for a fragment in Codestrates v2.

Cauldron uses the GoldenLayout layout manager to arrange editors and other components. The Cauldron authoring environment itself can be positioned either docked (see Figure 12.4), full screen, or floating as a window. Like the engine, Cauldron also builds on top of WPM, it is, however, not implemented using code fragments but regular script elements. To load Cauldron, a webstrate needs to include the WPM script and contain a bootstrap script that loads the execution engine using WPM. Cauldron does not need to be loaded through WPM until the user requests it to be opened. Only then is Cauldron installed transiently in the webstrate. Hence, the authoring environment does not clutter the memory of the browser when it is not needed.

### 12.5.1 Tree Browser, Inspector, and Tabbed Editor

The main purpose of Cauldron is to manage and author fragments. It provides three components that support this: the tree browser, the inspector, and tabbed editors.

The tree browser (see Figure 12.4B) shows an overview of the `<body>` element of a webstrate. Fragments inside the browser are indicated by icons for their type and can be opened by double-clicking on them. Regular DOM elements are displayed using folder icons, and transient elements are hidden in the tree browser. Using the context menu on elements, users can create new elements and fragments, move them, or delete them. It is, furthermore, possible to edit the HTML of elements that are not fragments (similar to the developer tools of a web browser). When editing the HTML of elements that are not fragments, changes are not automatically applied but instead have to be saved manually. Elements or fragments can also be dragged and dropped into the tree browser of other webstrates. The tree browser also allows viewing and uploading assets that are part of a webstrate. New assets can be uploaded using drag and drop.

---

**Figure 12.4**: An overview of the Cauldron authoring environment for Codestrates v2: Cauldron in its docked layout allows to see the application (A) to the side. Fragments and assets can be browsed in the Tree Browser (B) and inspected and renamed using the Inspector (C). Tabbed editors (D) allow to display code editors in a flexible way. The integrated console (E) allows to inspect logs without leaving Cauldron. (Cauldron can also be used full screen (which would hide A) or floating (which would put A full screen and Cauldron as a floating window above it.)

Once an element, fragment, or asset is selected in the tree browser, the inspector (see Figure 12.4C) displays attributes of the element like its ID or class, and allows to edit this information using the input field.

The tabbed editor (see Figure 12.4D), finally, shows editors of fragments or HTML elements, which can be displayed side by side in a flexible way. When editing fragments, the editor also contains buttons to run or auto-run the fragment. Using a collaboration layer, the tabbed editors also show the text cursor and selections of other clients that are currently editing a fragment.

### 12.5.2 Console, Toolbar, and Package Manager

Besides the main authoring tools, Cauldron also adds additional functionality to support development. The console (see Figure 12.4E) shows the current console output. This, for example, allows to also see the console output on browsers that do not have developer tools to do so — for example, on web browsers on mobile devices.

The toolbar of Cauldron adds various shortcuts to Cauldron and Webstrates functionality like changing the permissions of a webstrate, creating a new webstrate, duplicating a webstrate, or inspecting the
version history of a webstrate. It also allows to change the viewing mode of Cauldron (docked, full screen, floating window).

From the toolbar users can also open the Cauldron package manager. The package manager provides a user interface for the WPM component and allows to install or remove packages in a codestrate.

12.6 Examples

To illustrate the use of Codestrates v2 and Cauldron, we will present two examples of their use. The first example describes how to create a simple todo list application using Codestrates v2 and Cauldron, and the second example demonstrates how a computational notebook can be implemented.

12.6.1 Todo List Application

The first example is a simple todo list application\(^6\) that allows to add, mark as completed, and delete todos (see Figure 12.5). The example follows a simple MVC (Model-View-Controller) architecture and demonstrates its use in Codestrates v2 and Cauldron. The view consists of a single HTML fragment that renders the interface of the application, an external CSS library that is loaded using WPM, and a CSS fragment for styling (see Appendix B.1). A JSON fragment is used to store the data of the model and to synchronize the model with other clients. The controller, finally, is a JavaScript fragment that handles interactions with the view, and synchronizes its state with the model in the JSON fragment.

While this example is implemented using a MVC architecture, it is also possible to store the data of the todo items directly inside the DOM by using a persisted DOM element to implement the application instead of an HTML fragment. This would make the JSON fragment obsolete and instead combine view and model in the DOM.

12.6.2 Ganymede Computational Notebook

The second example is a Jupyter-inspired computational notebook called Ganymede,\(^7\) which allows to create code and markdown cells that can executed (see Figure 12.6). Code cells can be used to execute JavaScript code. Inside code cells the \texttt{print()} function can be used to output the result of a computation in the notebook. It is possible to run cells individually or all of them at once. Markdown cells can be edited by double-clicking on them, which displays a mark-

---

\(^6\) Todo List Example: https://demo.webstrates.net/codestrates-v2-todo-list/release/?copy (Retrieved February 22, 2022)

\(^7\) Ganymede Example: https://demo.webstrates.net/codestrates-v2-ganymede/release/?copy (Retrieved February 22, 2022)
down editor instead of a view. In our example, we also demonstrate how WPM can be used to load external libraries like Vega-Lite\(^8\) to display visualizations.

Technically, cells in the Ganymede notebook are code fragments that are stored in the DOM. Using the fragment API, Ganymede can run individual fragments and display the output of computations. Also the editors in the notebook are generated using the editor API from Codestrates v2—this also enables collaborative editing of cells. Our example, however, does not persist or synchronize the runtime state and outputs of the notebook. Excluding comments, Ganymede is implemented in less than 200 lines of JavaScript.

\section{Implementation}

Building on web technologies, WPM, Codestrates v2, and Cauldron are implemented using web technologies including JavaScript, HTML, and CSS. The code of all three components is available on GitHub.\(^9\) The following subsection describe each component in more detail.

\subsection{Webstrates Package Manager}

The Webstrates Package Manager is a single JavaScript script that implements the WPM object, which exposes functions to require packages. The script is usually located in the head element of the DOM and is executed on page load. Requiring packages from another webstrate is

\footnotesize
\begin{itemize}
\item \textbf{Vega-Lite}: https://vega.github.io/vega-lite/ (Retrieved February 22, 2022)
\item \textbf{WPM}: https://github.com/Webstrates/WPM
\item \textbf{Codestrates v2}: https://github.com/Webstrates/Codestrates-v2
\item \textbf{Cauldron}: https://github.com/Webstrates/Cauldron
\end{itemize}

(Retrieved February 22, 2022)
facilitated by using transclusion — similarly to the package manager of Codestrates v1 [52]. Using the custom tags described in Section 12.3, WPM scans the DOM for packages and parses their descriptor.

12.7.2 Execution Engine

Once a webstrate with Codestrates v2 is loaded, it first executes the WPM scripts, which then in turn executes the Codestrates v2 scripts. The engine of Codestrates v2 is managing the execution of fragments. To do this, the engine first searches the DOM for code fragments and adds mutation observers, which detect added or removed fragments. On load or when a fragment is added during runtime it is set up: The fragment is parsed and its type is detected. If the fragment is set to auto-run executable program code will be executed and markup or styling is rendered into a transient element. Table 12.1 shows the list of libraries used to execute the various fragment types.

The editors that Codestrates v2 provides for editing fragments are Ace, CodeMirror, and Monaco. Each of the editor types registers itself with the EditorManager, which manages the editor types and allows
<table>
<thead>
<tr>
<th>FRAGMENT</th>
<th>LIBRARY</th>
<th>WEBSITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>TypeScript</td>
<td>TypeScript</td>
<td><a href="https://www.typescriptlang.org">https://www.typescriptlang.org</a></td>
</tr>
<tr>
<td>Python</td>
<td>Brython</td>
<td><a href="https://brython.info">https://brython.info</a></td>
</tr>
<tr>
<td>Ruby</td>
<td>Opal</td>
<td><a href="https://opalrb.com">https://opalrb.com</a></td>
</tr>
<tr>
<td>Lua</td>
<td>Fengari</td>
<td><a href="https://fengari.io">https://fengari.io</a></td>
</tr>
<tr>
<td>JSON</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>HTML</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Markdown</td>
<td>Showdown</td>
<td><a href="http://showdownjs.com">http://showdownjs.com</a></td>
</tr>
<tr>
<td>LaTeX</td>
<td>LaTeX.js</td>
<td><a href="https://latex.js.org">https://latex.js.org</a></td>
</tr>
<tr>
<td>CSS</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>SCSS</td>
<td>Sass</td>
<td><a href="https://sass-lang.com">https://sass-lang.com</a></td>
</tr>
</tbody>
</table>

Table 12.1: The libraries used for the different fragment types. (Retrieved February 22, 2022)

to create editors. To create an editor, a fragment first has to be selected and parsed to the EditorManager. Once created, mutation observers in the fragment update the editor when remote changes happen in the code fragment and the editor updates the code fragment code once a user edits code in the editor.

12.7.3 Cauldron

Opposed to the engine, Cauldron is not opened and loaded by default when a webstrate is loaded. We decided to do that to improve performance on page load. Instead, Cauldron can be opened and loaded using an Edit button in the top right corner of the page.

Once loaded, Cauldron will initialize its interface, load the GoldenLayout library and setup the editors. By default Cauldron will be docked on the right side. The DOM of Cauldron is placed outside the body element of a webstrate in a transient element. By placing Cauldron outside the body, it does not interfere with application content in the body and allows users to use regular CSS styling rules without having to take Cauldron into account.

To store the current location of Cauldron window and its layout including all currently open editors, Cauldron uses the localStorage property of the browser. After a page refresh or page navigation, Cauldron retrieves this configuration from the localStorage and rearranges Cauldron in the same state it was before — except for scroll positions in editors, which are not stored in the current version.
In this report, we presented Codestrates v2, consisting of three components for the development of applications in Webstrates: the Webstrates Package Manager (WPM), the Execution Engine, and Cauldron. While WPM allows for easy transfers of functionality across webstrates, the engine adds an execution framework for code based on code fragments, and Cauldron an authoring environment for these fragments. Together, the three components provide a powerful platform for creating web applications in Webstrates.

ACKNOWLEDGMENTS

We would like to thank Germán Leiva for his helpful feedback. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 740548) and from Carlsbergfondet (grant agreement No CF17-0643).

APPENDIX

The appendix of this paper is included as Appendix B.
Computational media describes a vision of software, which, in contrast to application-centric software, is (1) malleable, so users can modify existing functionality, (2) computable, so users can run custom code, (3) distributable, so users can open documents across different devices, and (4) shareable, so users can easily share and collaborate on documents. Over the last ten years, the Webstrates and Codestrates projects aimed to realize this vision of computational media. Webstrates is a server application that synchronizes the DOM of websites. Codestrates builds on top of Webstrates and adds an authoring environment, which blurs the use and development of applications. Grounded in a chronology of the development of Webstrates and Codestrates, we present eight tensions that we needed to balance during their development. We use these tensions as an analytical lens in three case studies and a game challenge in which participants created games using Codestrates. We discuss the results of the game challenge based on these tensions and present key takeaways for six of them. Finally, we present six lessons learned from our endeavor to realize the vision of computational media, demonstrating the balancing act of weighing the vision against the pragmatics of implementing a working system.

13.1 INTRODUCTION

Software has today become synonymous with applications — bundles of functionality built for a specific domain and with a clear distinction...
between who the developer is and who the user is. Computational media is a vision of an alternative software model to the application-centric one. Software as computational media blurs the distinction between who is the developer and who is the user of software and the boundaries between application domains [78]. With traditional application-based software, we may use one tool to write notes, another tool to process data, a third tool to create a presentation, and a fourth to communicate with our colleagues. With software as computational media these four activities can happen in the same medium or in different depending on how the user combines their tools. Figure 13.1 shows how diSessa illustrates the distinction between software as applications and as computational media. He writes, “[m]onolithic, nonmodifiable applications have gaps and overlaps […] A computational medium allows seeding with small but extendible tools (black dots), but these tools can be organically enriched, altered, and combined, as successive layers here show” [77].

In our work, we study the potential of computational media as means for computer users to take control over their software and as a material foundation for the development of computational literacy. We see computational media as collections of information substrates [26]—software artifacts that combine content, computation, and interaction and can be treated as documents or tools depending on their use.

We have articulated four principles that should characterize modern computational media. It is malleable, so that users can adapt and repurpose their tools and documents in idiosyncratic ways. It is computable so users can execute arbitrary computations in any document and in any use situation. A modern computational medium must also provide the affordances we have come to expect from today’s software: that we can collaborate in real-time and use the many different devices we have at hand. Therefore, software should be distributable, so that tools and documents move easily across devices and platforms; and shareable, so that users can collaborate synchronously as well as asynchronously while using their own personal tools.

Webstrates [162] is a web-based prototype that since 2012 has served as a vehicle for us to explore the affordances, potentials, and consequences of software as computational media. It is based on a pragmatic attempt to create computational media by changing how web pages traditionally works: a webstrate (web + substrate) is a web page where any changes to its content, behavior, or appearance are automatically persisted on the Webstrates server and synchronized with all the users that are currently viewing it. Thus, any web page served by Webstrates can be collaboratively editable in real-time, including edits to program code embedded in the page. Klokmose et al. [162] demonstrated how

---

To begin with, we used the term Shareable Dynamic Media Klokmose et al. [162] as a play on Kay and Goldberg’s Personal Dynamic Media [149]. We later adopted the more straightforward Computational Media as introduced by diSessa and Abelson [78] for the same vision.
Webstrates supported the creation of software that allowed users not only to collaboratively author documents in real-time but also to collaboratively extend and reprogram their authoring tools while they were in use. Whether software served as a tool or as the object of a computing activity became a phenomenon of use instead of dictated by the software itself. Over the years, Webstrates has been used to explore these software qualities in the context of design [71, 143], scientific work [233], public libraries [119, 318], video editing [164], programming assignments [53], data analytics [16, 135, 210], collaborative affinity diagramming [191], and video conferencing [118].

While Webstrates demonstrably enables a number of use cases that are either extremely cumbersome or practically impossible to achieve with traditional software, the conflation of the space of using and developing software also introduces a number of principal, conceptual, or practical tensions. In this paper, we anecdotally present eight tensions between the vision and the practical realization of computational media that we have identified in the ten year process of creating Webstrates as well as making software with Webstrates. We present three short case studies illustrating how some of these tensions have materialized in concrete projects using Webstrates. Furthermore, we present a study of a three-week programming challenge using Codestrates v2 and Cauldron [49] — software development and code-authoring tools based on Webstrates. The tools are a product of compromises we have made in developing the tools to address the eight tensions, and we analyze the effect they have in actual use. We contribute a set of six lessons learned from our attempt to practically realize a vision of a new software medium in a concrete software platform that can be used to experience and study the potentials of the vision in practice:

L1: There is a critical difference between a system being technically reprogrammable and it being malleable to the user in praxis.

L2: Real-time collaboration is both a blessing and a curse, and particularly for collaborative programming; mechanisms to support
switching between synchronous and asynchronous collaboration in programming are essential.

L3: New concepts demand immediate and perceptible value for end-users to appropriate them and use them.

L4: Systems that are both real-time collaborative in use and development require a code execution model geared towards collaborative live programming.

L5: Technically competent users can get carried away by the vision and suspend their reflections on what is technically possible, which leads to frustration when their expectations are not met.

L6: Whatever user interface the users are provided will control the formation of their mental model of the whole system and its core principles, and users are quick to form conceptual blends with what is already familiar to build a working understanding, sometimes creating an uncanny-like effect when their understanding does not match the reality.

The paper is structured as follows: Section 13.2 positions our work towards related work on reprogrammable software and computational media. Section 13.3 summarizes the Webstrates and Codestrates platforms, and Section 13.4 presents a historical overview of their development. Section 13.5 describes eight tensions that have influenced the development of Webstrates and have been key concerns in the translation of vision to practice. Section 13.6 presents how tensions have materialized in three case studies. Section 13.7, Section 13.8, and Section 13.9 present the procedure and results of a study with twelve participants on a three-week programming challenge using Codestrates v2 and Cauldron. Lastly, Section 13.10 discusses our lessons learned in the process of realizing the vision of computational media.

13.2 RELATED WORK

The vision of computational media mainly originates from Kay’s first idea of a Dynabook [149]—a personal computer to support children in “learning by doing.” This idea evolved into a vision of personal computing that should accessible to anyone, which came with the challenge of supporting countless types of activities for all sorts of users: “The total range of possible users is so great that any attempt to specifically anticipate their needs in the design of the Dynabook would end in a disastrous feature-laden hodgepodge which would not be really suitable for anyone” [152].

This inspired Kay and Goldberg’s vision of personal dynamic media, a flexible software medium that would “allow ordinary users to casually and easily describe their desires for a specific tool” [152].
Kay and Goldberg compared such a malleable software to tangible materials like paper or clay, which are highly flexible and can serve unanticipated purposes when manipulated with the right tools. Other motivations that drove early research on computational media were to make software easy to learn and easy to program [78]. diSessa pointed out the tension between designing powerful systems, which can support unanticipated tasks, and their ease of use. He speculated that “a system composed of a large number of similar but subtly different structures is hard to learn and prompts mistakes and confusions” [76]. His vision aligned with that of Kay and Goldberg in that he argued for an integrated environment that served broad functionality by allowing users to modify it to their own needs rather than by offering a great number of pre-designed and specialized tools. Our vision of computational media builds on top of this vision and also emphasizes malleability and the possibility of users to modify their software to their idiosyncratic needs and combines it with a push for shareability and distributability as central aspects of modern computational media. These properties are essential in the current web of interconnected devices such as smartphones, tablets, laptops, and desktop computers, which stirs towards Weiser’s vision of ubiquitous computing [308].

Early attempts to realize such visions of highly flexible and personal software typically led to the design of programming environments or languages that were simple enough for novices to manage. diSessa explored the idea of a reconstructible computational medium [77] and created, together with Abelson, the Boxer platform [76, 78]. diSessa and Abelson [78] compared using Boxer to “moving around in a large two-dimensional space.” They applied a principled approach to designing Boxer, for example by employing principles such as spatial metaphors and naive realism. This way, Boxer aimed to leverage a person’s knowledge about space to navigate code in boxes. Boxes could be nested in each other, allowing to build structures of boxes to create applications, yet still making it possible to inspect and modify any code in nested boxes at runtime. Ingalls et al. [138] created Squeak based on the Smalltalk language which was used in the interim Dynabook prototype by Kay and Goldberg [152]. Squeak was a Smalltalk implementation whose virtual machine and programming tools were also written entirely in Smalltalk, which meant that all source code was available at all times and users could debug and change their programming environment in runtime with the same tools they used to develop end-user applications. The Lively Kernel system later lifted the idea of Smalltalk and Squeak to the web and made it possible to use the dynamic platform in a web browser [140]. Webstrates similarly uses the web as its foundation to pragmatically leverage its ubiquity, ecosystem of tools, and resources for learning. Compared to The Lively Kernel project, while sharing many similarities, Webstrates is built
around *shareability* and real-time collaboration as its core principles. Where The Lively Kernel introduces a new conceptual model for interactive components (based on Morphic [208]), Webstrates, in contrast, simply builds on the existing building blocks of the web.

Designing software so that it supports unanticipated needs has also been a core interest of the HCI and CSCW communities, often framed as “tailorable” software. Eagan et al. [85] argue that “there is a gap between the designers’ conception of how software will be used and its actual use,” and propose that users should be able to tailor their software to meet their needs as they change over time. For one example of such software, consider Robertson et al. [262] who implemented a system of modifiable buttons that “encapsulate appearance and behavior that is user tailorable. They are persistent objects and may store state relevant to the task they perform” [262]. MacLean et al. [197] also explored tailorable buttons, arguing that “tailoring should be a community effort,” and that for such systems to be successfully adopted, it was necessary to develop “a culture within which users feel in control of the system and in which tailoring is the norm.” Mackay’s study on MIT’s project Athena in the 1980s contributed empirical insights to that line of thought, finding that most users avoided customizing software “since time spent customizing is time spent not working” [201]. However, specialized workers in an organization may help their colleagues benefit from customizations: In the study, a small group of highly skilled users create and share customizations and a group of translators help less skilled workers adopt the customizations they need [199]. Mackay also observed that users not only *adapt* to software (i.e., learn how to use it) but also *adapt* it to their own needs, either by customizing it or appropriating it (i.e., repurposing it) for tasks other than the ones it was designed for [202]. These findings opened the door to designing software that is powerful by not only allowing tailorbility, but also by inspiring “appropriation” [81, 289], for example, by incorporating design qualities that support “unintended use” [263] or inviting users to add their own meanings to interface elements [117].

Tailorable software and computational media aim to blur the distinction between developing and using software, an idea that is also present in more recent explorations of *literate computing* [248]. Literate computing environments [223] are software environments where editable and executable code is interleaved with text, images and other rich media. The literate computing paradigm has become widespread in the data science community with the popularization of Jupyter Notebook [256] and similar computational notebook software (see an overview in Lau et al. [178]). They are often used for iterative and exploratory data analysis [267]. So-called no-code or low-code tools
such as Notion,\textsuperscript{2} Coda,\textsuperscript{3} or AirTable\textsuperscript{4} allow users to piece together personal software artifacts (e.g., spreadsheets, project timelines) or \textit{computational media} to serve their own idiosyncratic needs and goals \cite{105}. Codestrates \cite{257} brings the literate computing paradigm to Webstrates. Codestrates manifests the use of computational notebooks not just for data analysis and exploration but for developing interactive software. This blurs the distinction between using and developing an application as both happens in the same environment. However, the literate computing aspect may cause confusion and prevent people from using their prior programming knowledge, as it differs strongly from conventional code editors or IDEs \cite{50}.

An open challenge in exploring computational media is how to experimentally approach such a software model that is radically different from how today’s applications work. For instance, diSessa \cite{77} separated his work into three steps: First, a technological step that focuses on creating prototypes of a new type of medium. Second, a cognitive step that focuses on small-scale experiments with individuals or small groups of people. And third, a cultural step that focuses on how the medium impacts a community such as a primary education school. While diSessa was successful in the first two steps, the lack of sufficient funding made it impossible to realize the third. Our work on Webstrates and Codestrates, so far, similarly focused on the first two steps. The question of how computational media would be adopted on a large scale is neither new nor unique. Researchers have wrangled for decades with dynamic or computational media; however, most projects turned out to be short-lived, and only rarely do they escape the confines of academia, such as the case of Smalltalk, which has a small but lively community and implementations that are used in industry (e.g., GemStone,\textsuperscript{5} Pharo\textsuperscript{6}). We based our prototypes on web technologies, in part, hoping to sustain communities of developers and users, creating opportunities for studying the adoption of computational media on the web.

We are interested in eventually studying the adoption of computational media on a large scale, but we also believe it is still necessary to further understand users’ experience with such new types of software in detail. Edwards et al. \cite{87} reflected on users’ experiences with ad hoc interoperation of devices and services:

\begin{quote}
We believe that the primary challenges posed by our approach come from the user-experience perspective. In particular, the overarching question is whether users can accomplish their goals with a system that potentially provides less application sup-
\end{quote}

\textsuperscript{2} Notion: https://www.notion.so/ (Retrieved February 22, 2022)
\textsuperscript{3} Code: https://coda.io/ (Retrieved February 22, 2022)
\textsuperscript{4} AirTable: https://www.airtable.com/ (Retrieved February 22, 2022)
\textsuperscript{5} GemStone: https://gemtalksystems.com/ (Retrieved February 22, 2022)
\textsuperscript{6} Pharo: https://pharo.org/ (Retrieved February 22, 2022)
port than has been traditional. We believe that new UI techniques can help to maintain the usability that might be missing otherwise. (Edwards et al. [87])

While evaluating a platform’s success in terms of “accomplishing goals” and “usability” is important, this is only one form of user experience which assumes, for instance, that users are goal-oriented and emphasize usability. We are more interested in thinking of users as creative individuals who work through software to arrive at a place that was not anticipated in advance, simultaneously shaping and being shaped by the artifact. This perspective requires studies with a specific focus on how users understand computational media and how this understanding shapes the way they create software.

Finally, most of the research projects cited are dated and can be traced back to well before the current millennium. It is hard to imagine researchers reminiscing about word processors and image manipulation software from the 1970s and 1980s, yet that is what often happens with computational media. We are somewhat guilty of this ourselves. And while the past presents to us ideas and principles that are still valid, our work is motivated just as much by asking ourselves how computational media might look today. Are their characteristics fundamentally different from famous predecessors such as Boxer or Squeak? And if so, what principles lie under today’s computational medium, and what challenges lie in realizing it?

13.3 BACKGROUND: WEBSTRATES AND CODESTRATES

In this section, we present Webstrates, and the two iterations of our development and authoring environment Codecstrates v1 and v2. This section is based on previous work [49, 52, 162, 257] and describes what the platforms are. The next section will focus on the why and the motivations drove our prototyping of computational media through building platforms.

A basic understanding of the technical background and challenges in realizing the platform is necessary to understand some of the tensions (see Section 13.5) and results from the game challenge (see Section 13.7), that are based on technical limitations and challenges that has led to pragmatic design decisions.

13.3.1 Webstrates Overview

Webstrates is a platform we have used to explore the vision of computational media on the web. Technically, Webstrates is a web server. A web page served by Webstrates is called a webstrate: a “web substrate” that embodies content, computation, and interaction in the DOM (Document Object Model) of a page, including its HTML, CSS, and
JavaScript. Unlike a traditional collaborative application, where the client explicitly sends requests to the server to update its content or persist changes to a database, any change to the DOM of a webstrate is automatically persisted. For example, to create a todo list application where a user can add a new task to a list, a developer would traditionally have to program the server-side application that handles a request to create a new list (e.g., by persisting it to a database) and the client-side code that processes the response (e.g., adding the HTML for the new item). With Webstrates, the developer would only work on the client-side interaction for adding a new item, and the webstrate would automatically persist the changes. Moreover, Webstrates synchronizes the changes across all clients viewing it.

Authoring content in a webstrate can be done through tools that live in the webstrate itself, e.g., an “add new task” button next to a todo list, or by editing the DOM directly with a web browser’s developer tools. To create a simple shared notepad on a webstrate, a developer could open the developer tools, look for the `<body>` HTML element, and add the `contenteditable="true"` attribute. Then, any text typed on the `<body>` by an end-user would be persisted.

If multiple clients are viewing the same webstrate, local changes made by one are persisted and automatically synchronized with the others. Consistency between clients is achieved through operational transformations [89] on the document (see details in [162]). Each operation generates a new webstrate version, thus, users can access the history of changes of a webstrate. The operational transformation algorithm relies on a log of finely-granular operations, e.g., an insertion of a character, to give a sense of real-time collaboration, e.g., seeing another user type a paragraph.

Webstrates can be composed using the hypermedia principle of transclusion [226]. Transclusion is an important aspect of the compositability of documents. It strengthens both the malleability and shareability of webstrates. Figure 13.2 illustrates how a webstrate containing a SVG vector graphic representation of a tiger can be transcluded in both a document and a slideshow webstrate. Transclusion is realized using the conventional `<iframe>` element. Not only content like vector graphics can be transcluded into documents but also functionality and tools.

Besides acting as a persistence and synchronization layer, Webstrates provides a variety of APIs\(^7\) to support development with it. For instance, `transients` can be used to create elements and attributes that are not persisted and synchronized with the Webstrates server, `signaling` allows to send messages to other clients that visit a webstrates, `assets` allow to store files in a webstrate, and `permissions` allow to add read and write permissions on a per-webstrate level.

---

\(^7\) APIs were added to Webstrates over time and also after its initial paper [162]. Transients, signaling, and permissions, for instance, were added later and first published in the Codestrates v1 paper [257].
In summary, Webstrates serves as a vehicle to explore computational media on the web. It inherits the wide distributability of web pages, which can be easily addressed by URLs and accessed across devices. It supports malleability by allowing and persisting changes to content, interaction and computation on runtime; it enables shareability by synchronizing persisted changes across clients; and allows computability using scripts in the DOM.

13.3.2 Codestrates v1 Overview

Codestrates v1 and v2 are our efforts in creating authoring and development environments for Webstrates, built on Webstrates. Codestrates has undergone two iterations that we in this paper refer to as Codestrates v1 and Codestrates v2, which we summarize in this and the next section.

Codestrates v1 [257] took inspiration from computational notebooks such as Jupyter [256] and followed the literate computing paradigm [248], where authoring content as well as editing code happens in the same space. Unlike conventional computational notebooks that are primarily used for data analysis and visualization, Codestrates v1 allowed for creating application-like software such as the grocery list app shown in Figure 13.3. In Codestrates v1, the app is developed in a notebook format including its HTML content, JavaScript for behavior, and CSS for styling. “Deploying” an app is done simply by toggling full-screen on the so-called body paragraph containing the interface for the grocery list. At any time it is possible to break out of the interface to see and edit the implementation.

Codestrates v1 structures code in paragraphs: body paragraphs for HTML, style paragraphs for CSS, code paragraphs for JavaScript,
and data paragraphs for JSON. Paragraphs are displayed in a one-dimensional notebook-like view and can be grouped into sections. While the content of paragraphs is persisted, the user interface and editors are rendered transiently on each client.

Body paragraphs contain persistent DOM and display the DOM directly. DOM inside body paragraphs is editable in the view by using the `contentEditable` attribute. To edit a body paragraph as HTML, the DOM, first, needs to be serialized into HTML, can then be edited in a transient HTML editor in the Codestrates v1 UI, and is, finally, deserialized back into the DOM after finishing the editing. The deserialization process requires syntactically correct HTML and, thus, cannot happen live. This, in turn, prevented real-time collaborative editing of HTML in textual form in Codestrates v1. Because CSS of style paragraphs is stored in the text content of a single DOM element, changes to style paragraphs can be directly reflected in the DOM and applied by the web browser, allowing for real-time and collaborative editing of styles.

Code paragraphs can be executed manually or set to run-on-load, which executes them whenever the codestrate is visited. Codestrates v1 has a runtime environment for managing the execution of JavaScript in code paragraphs within a codestrate—a webstrate that runs Codestrates. Here, the order of execution could be controlled and scripts could import and execute other scripts within the same codestrate. Code paragraphs, further, can make use of data paragraphs to store data in the JSON format. A codestrate is self-contained, which means its implementation code is part of the notebook and can be edited in the same way as other paragraphs, which are used to implement an application like the grocery app.

Codestrates v1, furthermore, allows bundling functionality as named packages that could be shared between codestrates using a built-in package manager [52]. The package manager uses temporary transclusion to facilitate the moving of content between codestrates.
Figure 13.4: Data synchronization of HTML fragments between two clients: Fragment plain text content is synchronized both ways to the Webstrates server and Code Editor Instances. The rendered preview is also synchronized to fragment plain text changes, however, is not synchronized back to the fragment (reprinted from [49] with authors’ permission).

13.3.3 Codestrates v2 Overview

Codestrates v2 [49] is a generalization of Codestrates v1 and consists of three components: the Webstrates Package Manager (WPM), an execution engine, and the Cauldron IDE. Codestrates v2 has a similar goal to Codestrates v1 in that it aims to provide an execution engine for code, like for code paragraphs in Codestrates v1, and an authoring environment that runs directly in the web browser without setup. However, Codestrates v2 moves away from the strict computational notebook-like interface towards a more conventional authoring environment and, additionally, decomposes the execution engine, the authoring environment, and the package management.

Where Codestrates v1 presented code as paragraphs in a notebook editable through the user interface, Codestrates v2 introduces the concept of code fragments. Code fragments are reified code that can be programmmatically manipulated, executed, or edited through editors that are instantiable anywhere in the UI—for example, as cells in a notebook. A fragment is a `<code-fragment>` DOM element. Fragments are hidden and not rendered in the browser by default. The content of a fragment is stored as plain text in the text content of the fragment DOM element so that it is not interpreted by the browser on its own but can be managed by the execution engine. Codestrates v2 supports a variety of programming (JavaScript, Python, Ruby, and more), style sheet (CSS, SCSS), and markup languages (HTML, Markdown, LaTeX). Fragments with program code can be “run” manually to execute their code once or can be set to “auto-run,” i.e. being executed once when a page is (re-)loaded — similar to the “run-on-load” of paragraphs in Codestrates v1.

Fragments with markup as well as fragments with style sheets can only be set to auto-run and not be run manually. However, they are updated live and do not require a page refresh. Content from markup and style sheet fragments is rendered or preprocessed into
Figure 13.5: A screenshot of a web page with Cauldron docked to the right side of the window. The “edit” button in the left half of the screen — the application — can be used to open Cauldron.

a transient element. Content is synchronized from the fragment to the transient rendering so changes to the content of the fragment are automatically applied in the rendering (for markup and style sheet fragments). Because of this, users can collaboratively edit HTML fragments in real-time time without the conflicts that could happen in Webstrates or Codestrates v1. Changes in the rendering, however, are not synchronized back to the fragments (see Figure 13.4), as this could cause conflicts when trying to merge the changes back into the persisted fragment.

While in Codestrates v1 the authoring environment was tightly coupled to the execution engine, in Codestrates v2 fragments can be used and executed without an authoring environment, e.g., by using the browser’s developer tools. To author code directly in the web browser, however, Codestrates v2 also provides the authoring environment Cauldron, which enables users to create and edit code fragments. Cauldron’s UI is closer to the one of a regular code editor or IDE with a tree browser and tabbed editors (see Figure 13.5) rather than to a computational notebook. We did this to make the authoring environment more accessible to users unfamiliar with computational notebooks.

Codestrates v2 also comes with the more generalized Webstrates Packages Manager (WPM). Compared to Codestrates v1’s package manager, which only allowed to turn sections to packages, WPM allows to transfer any types of elements from the DOM as packages. WPM is
also used to update the implementation code of Codestrates v2 and Cauldron on each page load from a repository webstrate.

13.4 REALIZING THE VISION OF COMPUTATIONAL MEDIA

This section serves as a summary of motivations and design decisions when developing the Webstrates, Codestrates v1, and Codestrates v2 platforms. The section is structured into the three platforms and resembles a timeline of the development. Its focus is the why behind the development and how we used the platforms to explore the vision of computational media. Understanding the motivations and rational of design decisions is important for the understanding of some of the tensions in the next section.

13.4.1 Pre-Webstrates (2008–2011)

Webstrates and our computational media vision is not our first stab at these topics: The work sprung out of several attempts to rethink software particularly to support multi-surface interaction — interaction with computers that span multiple devices sequentially or in concert. We realized that the confinements of applications limited the possibilities of distributing work across multiple devices. The VIGO project [163] revisited the notions of application and document to propose a novel architecture for building interactive systems, based on instrumental interaction [25] and applied to an ad-hoc multi-surface environment. The idea of instrumental interaction conceptualizes software as dynamic collections of tools — not unlike diSessa’s characteristic of computational media. Shared Substance [113], further, proposed data-oriented programming as a way of expressing many of these same concepts in multi-surface environments.

In both cases, there were no existing, prior systems to bootstrap upon — everything had to be written from scratch. As such, applications were mostly proofs-of-concept viable for interactive and functional demonstrations but not necessarily for day-to-day use. The Scotty system [85] aimed to address this limitation by integrating instrumental interaction [25] concepts directly into the macOS Cocoa environment. As such, it could effectively bootstrap off of the existing environment, but was still limited by the different underlying design visions of traditional Cocoa applications and the instrumental interaction approach. The trade-off here is thus between being able to apply the research environment to day-to-day use and the purity of the research concepts and their resulting incongruity with the rest of the system.

Finally, with Webstrates [162], we took a compromise position between the two: it builds of existing web technologies and is able to draw on numerous existing web libraries, but bootstraps its own
system. This approach yields a compromise closer to the research goals associated with the computational media vision while still being able to draw upon the wealth of existing web systems.

13.4.2 Webstrates (2012–2016)

forming the vision (2012–2014). One of the realization that led to the development of Webstrates was that the distribution of a user interface between devices and collaboration between users were two sides of the same coin, and ideally should be supported by the same mechanisms. Furthermore, we realized that building software for multiple devices working in concert emphasized a need for software that could be changed, reconfigured, or even reprogrammed on the fly.

Our vision of computational media and Webstrates developed dialectically over three years. Initially, the first prototype of Webstrates was developed as part of an attempt to implement instrumental interaction [25] on the web inspired by the VIGO software architecture [163]. The idea was to use browser extensions as instruments to manipulate a persistent and real-time synchronizing web page, a “web substrate.” This would allow each user to have their own set of instruments installed in their browser, which would enable asymmetric collaboration between users with different tools and distribution of work across multiple devices.

The approach turned out to have a number of limitations: At that time, browser extension were only available in desktop browsers, which made the approach impossible on mobile devices. It enforced a more strict distinction between tool and document, or instrument and domain object, which were not compatible with the philosophy of instrumental interaction where an instrument can become an object and vice versa through use. The development of instruments and their use would happen in conceptually and practically different spaces and not through a general “clay of computing” as Kay and Goldberg [152] describe as the ideal for personal dynamic media. We wanted instruments to be able to manipulate other instruments in the same way as they would manipulate documents. This meant that instruments and objects, tools and documents, had to be made out of the same stuff. Our colleague, Beaudouin-Lafon, had made the theoretical insight that for instrumental interaction to work there would have to be information substrates that could serve as the material for creating both instruments and objects [26].

breakthrough (2014–2015). The big eureka moment was when we realized that the web that we used for prototyping was a hypermedia system and it supported transclusion. Transclusion on the web is implemented through the <iframe> element that allows for embedding one web page in another. When transcluder and transcludee
are served from the same domain, the security model of the browser allows the JavaScript runtime of the two pages to interact. That means that scripts from one of the pages can, e.g., manipulate the document of the other page and the other way around. We now realized that we did not need to implement our tools or instruments as browser extensions but could implement them as web pages as well. One of the first examples we built was a set of slideshow tools where the slideshow was its own webstrate and the slides could be edited and controlled from another webstrate through transclusion. Here changing a slide would simply be moving a CSS class attribute between slides in the slideshow webstrate. A more elaborate set of slide tools were presented in the 2015 Webstrates paper [162]. Now, whether what was being edited was considered application or document in the traditional sense started to dissolve and we felt we were getting closer to the “clay of computing” feel of software.

In the beginning, all development happened through the developer tools of the web browser. From those it is possible to inspect the DOM of a web page and edit its content including embedded JavaScript and CSS style sheets. It is also possible to execute arbitrary JavaScript code in the webpage from an interactive console. We quickly realized that the developer tools were not designed for development but rather debugging. The support for editing code was poor and did not provide any of the features — like code highlighting or linting — that developers were used to from IDEs. We, therefore, implemented a “code editor webstrate” [162] that allowed for editing embedded JavaScript and CSS of another webstrate through transclusion. At this point we also started having colleagues that started playing with Webstrates, and some core conceptual challenges started to emerge: A question we would often got was why our code editor webstrate could not edit the HTML of another webstrate but only scripts and styles. The explanation is that for live web page — that is web pages that are loaded in the browser — the HTML does not as such exist, only the runtime representation in shape of the DOM exist, and it is also that representation that Webstrates stores.

MATURING OF THE PLATFORM (2015–2016). After the first publication [162] and the initial public release of Webstrates in October 2015 we employed a full-time professional programmer to stabilize, improve, and document the codebase. In July 2016, we released a full rewrite of the codebase as a version 1.0, followed by the first set of APIs a month later: transients and signaling.

Both transients and signaling addressed shortcomings we had observed with the strict sharing model in Webstrates and were introduced to handle views, e.g., in a visualization library like D3.js, to avoid overloading the server by many rapid changes to the DOM.

8 D3.js: https://d3js.org/ (Retrieved February 22, 2022)
from such libraries, or to create local UI elements like shared cursors: Cursor positions are ephemeral information that can rapidly change and should not be persisted within a document. Transients allowed to create local cursor elements that are not persisted or shared, and signaling allowed to send information about the cursor positions between clients without having to write the ephemeral information into the document.

Given the increased use of Webstrates, a common request was the ability to edit a webstrate using a conventional IDE. To accommodate this, we developed Webstrates File System (WFS), a small service which developers could install on their computer that mounts a webstrate as HTML, CSS, and JavaScript files. Changes made to the files are automatically synchronized to the source webstrate.

13.4.3 **Codestrates v1 (2016–2019)**

**Literate computing with Codestrates v1 (2016–2017).** In the 2010s, computational notebooks—the most popular being Jupyter [166]—introduced the idea of literate computing [248] and found acclamation in the field of explorative data science [267]. Literate computing describes the idea of interweaving executable code with markup code in a notebook-like environment. Inspired by this idea, Codestrates v1 was born to explore how literate computing can be applied to the development of applications and not only data science, and to create a authoring environment native for Webstrates.

Codestrates v1 [257] was built on top of Webstrates and allowed to weave rich text and executable code within the same document, a codestrate. This blurred the distinction between development and use of applications as both the application and its code lived in the same continuous document. Besides combining development and use of applications in the same document, Codestrates v1 emphasized collaboration. Inheriting the distributability and shareability properties of Webstrates, it natively supported real-time collaboration within a codestrate and provided shared cursor and mouse positions of remote users. Codestrates v1, further, was the first platform to make extensive use of the transients API to generate non-synchronized user interfaces that allowed for more relaxed What-You-See-Is-What-I-See (WYSIWIG) [282].

We, later, extended Codestrates v1 by a package manager, Codestrate Packages [52], which added the functionality to easily share sections of a codestrate as packages with other codestrates. Codestrate Packages allowed to add and remove packages from a codestrate—here, every codestrate could act both as a repository where packages are retrieved from or as a document where the packages are used.

---

Codestrate Packages was motivated by more and more functionality that was put into Codestrates: shared cursors, a slide show application, a function to invite other users into a codestrate, and more. These features were always available in each codestrate and both cluttered the interface and reduced the performance. The package management was a way to store this functionality in a dedicated repository and only add those packages to one’s own codestrate that are actually needed. The package manager also allowed to create software by piecemeal without any programming: you could start out with a notebook, install a slideshow tool to turn the notes into a presentation, install a package to visualize data to put on a slide, install a video communication package, etc. This is what diSessa refers to as reconstruction [78].

codestrates v1 in use (2018–2019). Over the years, Codestrates v1 transformed from a research prototype to a platform for creating applications in Webstrates. We and other colleagues used Codestrates v1 in a diverse range of projects: In Vistrates [16], we used Codestrates v1 to create a collaborative and modular visualization tool. In the development of Vistrates, the UI architecture of Codestrates v1 served both as leverage but also limiting: Things were easy when following the Codestrates v1 structure of sections and paragraphs, but more difficult when trying to diverge from it. For instance, changing the layout from a one-dimensional notebook layout into watching paragraphs side-by-side, would have required to implement a complete layout engine. The structure of sections and paragraphs, similarly, did not allow further nesting of sections and allowed for only two hierarchical levels of content.

When using Codestrates v1 for collaborative programming assignments in a university course [53], we found the notebook-like structure of documents to be limiting in some situations and to cause a lot of scrolling. In some instances, students also made codestrates inoperable by accidentally removing the implementation of Codestrates v1 itself from the document or hiding all elements—including the UI—with their CSS styles. This was possible as the implementation of Codestrates v1 and its user interface lived in the same space as the application or content a user is working on. This means that, for instance, styles applied to an application, e.g. changing the background of all <div> elements, was also applied to the UI of Codestrates v1. And in the same way did removing all <div> elements from the body, for example, to remove all items in a todo list, did also remove the implementation of Codestrates v1 itself, leaving the user with a blank page. Still, the linear structure supported students in thinking about assignments one paragraph at a time, and the shareability and real-time collaboration were clear benefits in that study as it allowed students to avoid conflicts like in other version-control systems like Git [53].
In another project together with a group of nanoscientists, we used Codestrates v1 to create a prototype of a computational labbook [233] (see Section 13.6 for more detail). In it, we wanted to reuse existing scripts of the researchers, this, however, was not easily possible as they used Perl and Python as their primary programming languages. Codestrates v1, however, only supported JavaScript as a programming language, making it difficult to integrate the scripts of the researchers with our prototype. For our prototype we therefore had to employ an additional server tool, that were executing Perl and Python scripts that were sent from a codestrate using REST and WebSockets. Videostrates [164], a collaborative video editing prototype built on Webstrates and Codestrates v1, similarly required a high performance server to compute video streams as this was not possible with the limited performance of the web browser.

During our increasing use of Webstrates and Codestrates as a prototyping platform in research projects where prototypes were deployed in the wild, we experienced that users’ browser extensions (like, e.g., Grammarly) would pollute documents with annotations or, even worse, break a webstrate by removing or editing critical parts of the document. Furthermore, libraries and frameworks we would use were built for traditional web development and not designed with the anticipation of the DOM being persisted. For instance, Monaco\(^\text{10}\) that we used for code editing, would write to the document in ways we had difficulties controlling. We therefore resorted to develop an optional protected mode for a webstrate that would ensure that edits from browser extensions and third-party libraries and frameworks would not be persisted in the DOM. However, this had the trade-off that edits that should be persisted required a certain option added to the method called when manipulating the DOM from JavaScript.

13.4.4 Codestrates v2 (2019–2020)

Decoupling and Generalization (2019–2020). Inspired by aforementioned shortcomings of the initial version Codestrates v1, we took a step back and reconsidered the overall structure of Codestrates v1 and what made a codestrate a codestrate. While reflecting over what defines the core of Codestrates v1, we found that Codestrates v1 could be divided into three parts: (1) its computational model of paragraphs and their execution, (2) the notebook-like UI environment consisting of paragraphs and sections, and (3) the package manager for sharing code between notebooks. As all parts were tightly glued together and one required the other to function, we decided that a first step would be to separate the three. We, therefore, decoupled the computational aspect of paragraphs with their

---

\(^{10}\) Monaco: https://microsoft.github.io/monaco-editor/  
(Retrieved February 22, 2022)
representation in the user interface. This would allow computation to behave in the same way independently from the view. Further, this would allow to display these computational units in different views, e.g., using a notebook-like environment, a conventional tabbed IDE interface, or visual programming interfaces where computational units are placed on a 2D grid. Finally, we implemented a general purpose package management system for Webstrates. The resulting architecture of components in Codestrates v2 \cite{49} consisted of the Webstrates Package Management (WPM), the execution engine, and the Cauldron authoring environment.

**From Paragraphs to Fragments (2020).** We renamed the “computational units” in a document from paragraphs to *fragments* as naming the computational units as “paragraphs” in Codestrates v1 was related to its notebook-like interface. The computational engine of Codestrates v2, thus, manages the execution of fragments, the creation of virtual machines for executing code, as well as providing functionality to reuse code from other fragments.

Fragments could be placed anywhere in a document/in the DOM and structured arbitrarily, introducing the possibility to create deeper hierarchies (e.g., using `<div>` elements as folder). Besides this structural flexibility, it further allowed us to create different editors for editing the content of these fragments: It was then possible to view fragments both in a one-dimensional notebook representation as before, but also in a tabbed editor, that is, for example, found in usual IDEs or a two-dimensional canvas where fragments can be freely moved around.

Editors for fragments can be instantiated and placed anywhere in the user interface, and multiple can simultaneously show the same fragment. This means that it is, e.g., possible to put live editable code on a slide in a slideshow.

**Separating the Authoring Environment (2020).** While the fragment and editor engine provides helper functions to instantiate code editors to edit fragments, it does not create them by itself and instead requires an authoring environment to browse, view, and edit fragments. To demonstrate this, we created the authoring environment Cauldron, which allowed to browse paragraphs in a tree browser and to view and edit paragraphs in a tabbed editor. We decided to create a more conventional authoring environment over, e.g., a notebook-like environment as in Codestrates v1, because our goals shifted from exploring the computational notebooks as a development environment towards investigating malleability and collaboration in using when using a computational medium. Therefore, Codestrates v2 and Cauldron would act more as a platform for further projects, rather than being a research prototype in and by itself. Providing users with
13.5 TENSIONS IN THE REALIZATION OF COMPUTATIONAL MEDIA

While revisiting and analyzing the development history of Webstrates and the two iterations of Codestrates, we were able to identify eight tensions that influenced design decisions during the development. We refer to “tensions” as combinations of two opposing viewpoints that pull against each other, opening a spectrum in between them. Tensions are also related to differences between the principles of our vision and the practical hurdles involved in order to create a working system with the limited resources at hand. For example, a tension between how malleable a system is compared to how stable it is. Tensions, therefore, required us to find trade-offs between the opposing viewpoints. Here, we briefly summarize each of these tensions. The remainder of this section illustrates them through design decisions made in the development of Webstrates and Codestrates.

T1: Malleability vs. Stability is the trade-off between how much the user can adapt the environment to their own needs and how much the user can break it. T2: What is Shared vs. What is Not Shared focuses on the complexity of managing shared, local, or personal content in a sharable medium. T3: Editing Directly vs. Editing Indirectly describes how directly the user interacts with the computational medium or whether those interactions are mediated by other concepts. T4: Big vs. Small Distance Between Development and Use Views deals with how distinct development is from typical use. T5: Authoring Environment Developed in Itself vs. Not Developed in Itself similarly deals with whether the devel-

---

11 To load it with the authoring environment enable one would then have to append `?edit` to the URL.
opment environment is bootstrapped within itself. *T6: Self-contained vs. Auto-updating Authoring Environment* focuses on how independent a computational medium is, from a self-contained entity to deeply interlinked with dependencies on other computational media. *T7: Creating Something New vs. Offering Something Familiar* raises the trade-off between creating a new, “pure” implementation from first principles and drawing on the existing and the familiar. Finally, *T8: Working With Your Own Tools vs. Adopting Built-in Tools* deals with the challenges of using custom tools or existing ones in the editing environment.

13.5.1  *T1: Malleability vs. Stability*

One key principle of computational media is to be malleable. Thus, tools created with Webstrates should be able to be changed by users, allowing users to tailor their tools in idiosyncratic ways. Malleability, however, does not only grant users the power to modify tools for good but also for ill: Erroneous modifications can lead to problems ranging from unexpected interactive behavior to broken functionality to loss of data. This is in stark contrast to the rigid and non-malleable applications we know. These are usually more stable as their functionality cannot be changed and always behaves in exactly the same way.

Webstrates started as a very malleable, yet unstable, platform. Using primarily the web browsers’ developer tools to modify a webstrate, everything within a webstrate could be changed by users. While this provided power, it also permitted to easily break applications, e.g., by accidentally deleting the wrong elements from the DOM and thereby removing essential parts of an application.

Codestrates v1 shifted the platform to more stability: documents are no longer edited without constraints in the developer tools but within the web page in dedicated paragraphs. This prevented accidentally deleting DOM elements while editing code. Code within paragraphs, however, is still highly malleable and has a global scope, allowing users to modify existing parts of applications or adding new functionality. This global scope, on the other hand, also allows the code of an application built with Codestrates v1 to also affect the Codestrates v1 platform itself, potentially corrupting the platform with faulty code.

In Codestrates v2, we intentionally increased the separation between the authoring environment and the content produced by users. Technically, we exploited that it is possible to move content outside of the `<body>` element so the editor, Cauldron, was not part of what is typically considered the content of a web page. This prevented, for example, some CSS styles to affect the Cauldron editor. The implementation code of Cauldron is, furthermore, loaded from another webstrate on page load in Codestrates v2, which, again, made the platform more stable as the implementation code of Cauldron could
not easily be corrupted. At the same time, this impaired malleability as modifications of the Cauldron environment were not easily possible from within the document. Lastly, by employing the protected mode by default, Codestrates v2 prevented elements that were not explicitly approved by programmers from being persisted on the server, adding stability. However, this also adds to the complexity of the mental model of Webstrates, which no longer is as simple as all changes to the DOM being persisted equally.

13.5.2 T2: What is Shared vs. What is Not Shared

When collaborating in Webstrates, there are different levels in which the state of an application can be shared. On the one end of the spectrum, everything within the application is shared with other clients: UI, elements, what is visible on the screen, up to system state. On the other extreme, nothing is shared with other clients and users work independent of each other, e.g., by using transient elements, the protected mode, or regular websites that are not synchronized with a server. Webstrates initially began by sharing the whole DOM. Later, Codestrates v1 and Codestrates v2 added features to allow local and not persisted DOM elements, e.g., in transient elements that allow for more relaxed WYSIWIS (What-You-See-Is-What-I-See) interfaces.

In the initial version of Webstrates, every element in the DOM was persisted with the server. To have different views on documents, one needed to transclude “document-webstrates” into different types of “editor-webstrates.” While the DOM was shared, the only part of an application that was local, was the JavaScript state. This allowed to create shareable and collaborative applications by only working in the front-end, using the DOM as a shared and persisted layer of information.

Codestrates v1 moved towards sharing less: By using transient elements that were not persisted on the server, parts of applications and the user interface could be local to clients. This was especially useful for local user interface elements such as dialog boxes or menus, that if shared among all clients, would cause disruption for during real-time collaboration. Which paragraphs and sections are visible, however, was still shared across all clients. This caused problems such as jumping content when collaborators opened or closed paragraphs during real-time collaboration, e.g., in [53]—similar problems can occur in collaborative writing tasks in online tools like Google Docs or Overleaf.

While sharing elements was the default and transient elements had to be used explicitly to prevent this in Codestrates v1, Codestrates v2 moved towards the default of not sharing elements by using the protected mode in Webstrates. This prevented the pollution of the persisted DOM with elements that are, for instance, automatically
created when using some JavaScript libraries, and, contrary to the shared view of paragraphs, allowed users to view different code fragments in each client.

13.5.3 T3: Editing Directly vs. Editing Indirectly

In a malleable computational medium, the user can edit content, but that editing can be direct or indirect. In direct editing, the user acts on the objects she perceives to effectuate a change, such as when editing a traditional web page’s DOM through a browser’s developer tools. In indirect editing, that change is mediated through some sort of a process, such as when editing an HTML file and reloading it to see the changes. Of course, in virtually all systems, there is some level of indirection. Even in the example above, the developer tool presents a mediated view of its in-memory representation of the DOM. When the user edits that representation, the browser transforms that edited representation into the the appropriate changes to apply to its internal data structures and reflects those in both the rendered web page and the developer view. Nonetheless, to the user, that edit would be perceived as acting (more) directly on the DOM, while editing the HTML file would be perceived as more indirect.

Basman et al. [21] refer to this as divergence in software systems: the difference between the representation that software has when it is running and the representation it is being edited in. Sometimes the divergence is almost imperceptible, e.g., when editing CSS rules that are immediately parsed and applied. The divergence is, however, more clear when editing imperative code that needs to be manually executed, and programmers quickly learn about runtime state where there is a stack. Normally this runtime is torn down and recreated when the code changes but in Webstrates the JavaScript runtime should ideally persist and apply changes directly.

In the different iterations of Webstrates and Codestrates, we explored different ways of modifying an application. In the original Webstrates, users could modify the DOM through the browser’s developer tools DOM editor or by typing in a contentEditable element (which supports a limited set of in-place rich text editing operations within a rendered web page). In both of these cases, this editing is perceived as direct, with the system transparently mediating the user’s expression of those edits and their updates to the DOM. Those changes to the DOM were then synchronized to the Webstrates server. Conflicts, however, could occur, especially in the developer tools, where edits occur on a per-element granularity. As such, edits to a DOM element are only propagated to the DOM — and hence to the server and other clients — then the user has finished editing that element.

Codestrates v1 introduced a way to add editable elements to a webstrate through body paragraphs. As in Webstrates, a user could
edit a webstrate in two ways: (1) directly in the paragraph using the contenteditable attribute and (2) using a HTML editor, but in Codestrates v1 this HTML editor was built into the Codestrates environment. A user would no longer need to leave the environment to edit the webstrate. This HTML editor works similarly to the developer tools, serializing the DOM of a body paragraph into text that can be edited and “saved,” and deserializing the changes back into the DOM (although with less integration and robust tracking to, say, maintain event listeners of unedited elements in the paragraph). Due to technical challenges, the editor was not shared and, thus, like in the developer tools, conflicts could occur when multiple users edited the same body paragraph at the same time.

In Codestrates v2, the HTML editor modifies HTML fragments, backed as plain-text elements in the DOM. This change reduces conflicts as edits are now synchronized at a keystroke level of granularity rather than at the Codestrates v1 paragraph-level. However, it also adds a degree of indirection, since changes to that text element are then rendered into a transient DOM element (see Figure 13.4). As a consequence, the user does not actually modify the DOM, rather the DOM is a projection of the rendered plain-text element. If they were to modify that rendered DOM element in the developer tools, those changes would not be synchronized to other Webstrates clients and would be overwritten the next time to the backing element was re-rendered.

13.5.4 T4: Big vs. Small Distance Between Development and Use Views

The development and the use of applications is usually separated, e.g., using an IDE for the development and the web browser for the use. Switching from one context to the other can involve shifting focus from one part of the screen to another part of the screen, from one application to another, or from one device to another (e.g., when testing mobile apps developed on the computer on a phone). We call this the distance between the development and use views of software. The distance is dependent on the time required to switch from one view to the other and the effort it takes to switch to the development view.

Computational media and Webstrates allow to bring the two views closer together and reduce this distance by using webstrates that inhabit both content, computation and authoring environment of an application in the same document. Initially in Webstrates, editing usually happened either in the developer tools or a separate “editor-webstrate” that transcluded the webstrate to be edited. This required a back and forth between different views when developing. While this distance was smaller than when using conventional development tools, it still provided a clear separation between the use and the development views.
Following the literate computing paradigm [248], Codestrates v1 weaved the development environment into the same space as the representation of the application. This avoided the need to switch between different views and narrowed the distance between development and use. Literate computing, however, is an uncommon model for developing applications and therefore it was sometimes difficult to mentally separate what is part of the development environment and what is part of the application [50].

With the introduction of Cauldron in Codestrates v2, we reverted the development model back to a classic separation of development and use by increasing the distance again. This was partly to reduce the confusion that resulted from the interweaved development environment in Codestrates v1. In Codestrates v2, the Cauldron environment is separated from the application—like the web browser developer tools in Webstrates. While the distance is similar to the one of the developer tools in Webstrates, Cauldron allows to be independent of the web browser and to provide targeted features for fragments. Codestrates v2 allows creating software where the distance is as low as in Codestrates v1, e.g., the simple Jupyter-notebook clone Ganymede [49].

13.5.5 T5: Authoring Environment Developed in Itself vs. Not Developed in Itself

Authoring environments can be either developed in themselves or not. When developed in itself, an authoring environment’s implementation is part of itself and is bootstrapped from within—similar to systems like Squeak [138] or Pharo. Such an authoring environment allows users to change core behavior of this very authoring environment from within itself—without requiring additional development tools like an external IDE. While this empowers users to change even core behavior of their applications without the need of external software, it also brings the risk of enabling users to break core features of their software.

Codestrates v1 was the first platform that added an authoring environment to Webstrates that was written in itself: The implementation of Codestrates v1 was written in the same types of paragraphs and sections as applications implemented with it. A bootstrap script was used to kick off the environment on page load. As the implementation of Codestrates v1 was part of the documents themselves, it provided means to users to change any aspect of the authoring environment without the need for any external tools, as, for example, done in systems like Vistrates [16], the computational labbook [233], or a collaborative writing editor [50].

In Codestrates v2 the authoring environment Cauldron was implemented using an external IDE and updated on the Webstrates server using custom build scripts. This step was motivated by the way
Codestrates v2 was developed compared to Codestrates v1: While Codestrates v1 was developed by the researchers themselves, Codestrates v2 was primarily developed by two professional developers, which relied on using their known tools and workflow in order to work efficiently. While making the development process faster and more efficient, this came at the cost of the ability to change the authoring environment from within the system, as Codestrates v2 and Cauldron were not implemented using code fragments that are otherwise used to implement applications in Codestrates v2.

According to Justice [145], Smalltalk and HyperCard are two other examples of platforms for modifiable software on this spectrum. Smalltalk lets the user reprogram everything down to redefining the `+` operator, whereas in HyperCard the software environment itself is not written in the same language — HyperTalk — as the user software.

13.5.6 To: Self-contained vs. Auto-updating Authoring Environment

The authoring environment of a document can be either self-contained, i.e., its implementation code is part of the persisted part of a document and was cloned from a prototype document, or automatically loaded (and updated) into a transient element from another webstrate or repository on page load. This behavior is similar to copy-by-value (the first case) and copy-by-reference (the second case). While being self-contained makes it easier to experiment with modifications on the authoring environment, being loaded from another webstrate allows to easily distribute updates to all clients. Whether the authoring environment is self-contained or auto-updating is unrelated to where it was developed: A self-contained authoring environment can be not developed in itself and an auto-updating one can be developed in itself.

In Codestrates v1, the authoring environment was self-contained in each codestrate. This made it possible that the authoring environment could be modified locally and modifications were persisted on the server. However, when code of the authoring environment changed, every codestrate individually had to be updated with the new code.

Codestrates v2 shifted the focus on being more easily updateable: Both the implementation of Codestrates v2 and the authoring environment Cauldron were by default transiently loaded from another repository webstrate on each page load using WPM. Thus, once the repository webstrate was updated, it only required a page reload to update individual codestrates. While this made rolling out updates easier, modifying the implementation of Codestrates v2 was more difficult, as modifications to the authoring environment were lost after each page load. To mitigate this, a second mode was added to the WPM that loads packages (like the authoring environment) in a persisted way into a webstrate.
13.5.7  \textit{T7: Creating Something New vs. Offering Something Familiar}

By building on existing web technologies, Webstrates allows developers to reuse existing programming knowledge and a vast selection of JavaScript libraries. This can, however, cause problems when some aspects of the development or libraries are different from the expectations of a user. For example, some libraries “consume” elements of the DOM when initializing themselves, which prevents the correct initialization after the page was loaded once. But also developers have to rethink some of their development practices—especially if they developed for the “normal” web before—as Webstrates meddling with some of the fundamentals of web development such as creating persisted web application without providing an explicit backend.\textsuperscript{12}

In the initial version of Webstrates, conflicts with existing libraries could primarily happen when libraries modified the DOM in a destructive way or when polluting the DOM: JavaScript libraries often expect a “clean” DOM on page load. This, however, clashes with Webstrates’ persisted DOM and causes DOM elements to persist on the server that are not supposed to be persisted, for example, visualizations using Vega.\textsuperscript{13} After a page reload, the DOM elements of these visualizations are still in the DOM, the JavaScript state, however, is missing, causing the visualizations to be not interactive anymore. Similarly, libraries often do not listen for changes in the data in the DOM, as the DOM is usually static, whereas in Webstrates other clients can change elements—in order for these libraries to work collaboratively, manual modifications need to be implemented to make them work as expected. Also using the developer tools of a web browser as the primary editor was an uncommon way of developing applications.

In Codestrates v1, most of the library issues from Webstrates persisted and were even increased by interference with the authoring environment in the DOM: When using the Bootstrap CSS library, for example, the CSS styles would also interfere the representation of the authoring environment, causing various display errors like wrong fonts, colors, or margins. The introduction of transient elements, however, made it possible to manually adapt libraries to Codestrates v1, e.g., by rendering Vega-Lite\textsuperscript{14} visualizations inside a transient element instead of directly in the DOM. For users, the unconventional way of programming in a notebook environment could cause confusion: Being able to run JavaScript code inside an application that is already running is contrary to how web applications are usually developed. But also the introduction of the “auto-run” of paragraphs caused confusion, e.g., when users were not sure how to execute something after the page was loaded \[50\].

\textsuperscript{12} While the Webstrates server is technically the backend, development of applications in Webstrate usually only happens in the front-end.
\textsuperscript{13} Vega: https://vega.github.io/vega/ (Retrieved February 22, 2022)
\textsuperscript{14} Vega-Lite: https://vega.github.io/vega-lite/ (Retrieved February 22, 2022)
13.5 TENSIONS IN THE REALIZATION OF COMPUTATIONAL MEDIA

Codestrates v2 mitigated various of the library-related problems: By introducing the protected mode, DOM elements created by libraries would by default not be persisted on the DOM, and by moving the authoring environment outside the `<body>` element, it was not affected by some CSS styles that target the body anymore. However, using the protected mode also requires the creation of persisted elements to be made explicit: if a developer forgets to do so, the created elements are discarded after the next reload and/or cause confusion as they are not persisted to other clients. For users, Codestrates v2 aimed to create a more familiar environment with tabbed editors and a regular tree browser for code fragments — allowing users to reuse their experience with other code editors or IDEs.

13.5.8 *T8: Working With Your Own Tools vs. Adopting Built-in Tools*

To modify a malleable medium users require an authoring environment. This can be either a user’s own tool that connects to the medium, like a code editor or IDE, or a built-in tool. Especially for programming activities it can be desirable to integrate the medium with existing tools, as this allows users to continue to use their familiar tools and workflows. On the other hand, not every user has an established workflow or existing tools. For these users it can be useful to provide a built-in authoring environment, which can be used without any setup.

Webstrates came without a built-in authoring environment, instead users had to either use the developer tools of their web browser or the Webstrates File System (WFS) utility. Using the developer tools introduced various challenges: The developer tools are intended to debug but not develop applications, thus, editing code using the developer tools had drawbacks such as a lack of code highlighting. Furthermore, some browsers, e.g., on mobile devices, do not even provide developer tools for users. WFS on the other hand allowed users to use their local code editor or IDE to modify a webstrate. This, however, required user to setup this local development environment themselves, making this a utility more tailored towards advanced users.

In Codestrates v1, the authoring environment was a central part of the document and accessible right within the web browser. This made the environment independent of devices or web browsers and allowed users to collaboratively author documents. To manage running individual code paragraphs and preventing some to run on load, Codestrates v1 changed how scripts are stored in the DOM: instead of in `<script>` tags they were stored in `<pre>` tags, rendering some functionality of WFS useless, which allowed to mount `<script>` elements as files.

While changing the executable code from paragraphs to fragments, Codestrates v2 continued to have the same incompatibility with WFS,
The built-in authoring environment over their own tools. Even though users can copy code both in Codestrates v1 and Codestrates v2 back and forth into their local code editor, this causes overhead and is inconvenient. Still, for some users, as, e.g., seen in [50], this was worthwhile as their local editor allowed them to use more advanced features like refactoring or an advanced search, as well as their personalized shortcuts.

13.5.9 Discussion of Tensions

Many of these tensions fall out of the overarching trade-off of what we refer to in the title as principle vs. pragmatism, although it could also be expressed as conceptual simplicity or conceptual purity vs. realism. For example, T1: Malleability vs. Stability manifests this challenge in the difficulty of finding the right balance between giving the user lots of rope to work with, but with the risk that they can easily tie themselves in knots, similar to the way that Smalltalk users must make frequent backups of their environment, especially when, say, redefining the + method for all objects or how strings appear on the screen.

In T2: What is Shared vs. What is Not Shared, it reveals itself through the evolution from the conceptually simple sharing model that what is shared is “The DOM, the whole DOM, and nothing but the DOM.” This purely shared medium needed to evolve to handle the reality that locally but not globally-pertinent content still needs to be expressed through the DOM—hence the introduction of the notions of transients and signals.

In T3: Editing Directly vs. Editing Indirectly and T5: Authoring Environment Developed in Itself vs. Not Developed in Itself, the challenges of bootstrapping a computational medium run into the needs for the medium to be editable within itself. While it is today necessary to balance these different needs, one could imagine that some day the tools become sufficiently rich and mature as to no longer need such external tools—much as many programming languages eventually reach sufficient maturity to no longer depend on other programming languages to create them.

Finally, in T7: Creating Something New vs. Offering Something Familiar and T8: Working With Your Own Tools vs. Adopting Built-in Tools, novel principles run against the recognition that users are not a blank slate; they have baggage from their prior experiences with other environments and assumptions based on their different principles. Some mechanism is thus necessary to help users unlearn what they have learned or otherwise break with those prior assumptions.
13.6 CASE STUDIES

In the following we will present short case studies where we have used Webstrates in three different projects to both illustrate a breadth of possibilities and to discuss how some of the above tensions have materialized in the projects.

13.6.1 Computational Labbook

We collaborated with a group of nanoscientists to explore how their use of laboratory notebooks could be remediated as computational media [233] using Webstrates and Codestrates v1. Through a series of participatory design activities over the course of two years, we built a prototype to support their computational design of macromolecules — what the scientists referred to as the “dry” part of their work. In the labbook they could create a computational workflow by drag-and-dropping various tools from a tool panel, and mix it with written notes. Figure 13.6 shows a notebook where an ASCII-based pattern editor for RNA structures have been added together with a tool for visualizing the structure in 3D. We observed how most of the scientists saw dealing with scripts and code as a necessary but secondary aspect of their work. So, contrary to traditional computational notebooks (such as Jupyter) code was not made front and center. The implementation of the various tools was accessible and editable, but in ordinary use, a computational workflow was created using drag-and-drop.

The computational notebook aggregated a number of scripts and tools the scientists otherwise would use through the command-line and multiple applications. Each notebook was a self-contained computational environment that they could configure for a particular experiment, share with colleagues through its URL, and interact with and edit collaboratively. Some of the tools would rely on distributed computing, and execute computationally heavy scripts on a dedicated server. This reduced the scientists need for maintaining a complex computational environment locally on their computer.

The labbook case addresses tension T1 (Malleability vs. Stability). Stability is provided with a set of tools, or building blocks, that can be used to create new computational workflows. What diSessa and Abelson [78] would refer to as reconstruction. However, the scripts embedded in the tools are reprogrammable from within the labbook. In day to day work, the notebook can be used as a regular piece of software, but in the critical situation where reprogramming is needed, it is possible. We intentionally increased the distance between development and use views (T4: Big vs. Small Distance Between Development and Use Views) as programming was seen as a necessary but not primary activity by the scientists. Being able to adjust this distance for particular situations of use, we see as a critical aspect
of computational media. Each labbook contained all scripts (T6: Self-contained vs. Auto-updating Authoring Environment). This meant that changing a script in one labbook would not have an effect on others — for good and for ill. That changes to a script do not propagate when they, e.g., are done to accommodate a particular experiment is meaningful, but if they are done to fix a general bug such as a memory leak, it is not. Hence, computational media requires mechanisms for managing this tension, of which we yet only have implemented rudimentary support for expert users with our Webstrates Package Manager.

13.6.2 Building Prototypes for Public Libraries

In the PLACED project [119, 316, 318], we collaborated with public libraries in Sweden, France, and Denmark on how to digitally support and document events that happened in the libraries. This included events such as book readings, crafts workshops, talks, courses, and more. The libraries struggled often ad-hoc relied on social media pages, and struggled with documenting what had happened, been discussed, or produced during the activities. In this project we used Webstrates for rapid, exploratory, and iterative prototyping of new tools for both visitors to the libraries and organizers of events.

We built a prototype, PARTICIPATE, where events were represented with what we called an activity sheet (see Figure 13.7). Activity sheets are inspired by event pages on social media such as on Facebook. Participants can share posts, images, and access books in the library related to the event. Organizers can configure the activity sheet, enable Q&As, setup a bookshelf, and more. In many ways PARTICIPATE is a
conventional web app and does not exhibit characteristics of computational media. However, the Webstrates platform allowed us to design high-fidelity functional prototypes with data persistence and real-time collaboration without the need for backend development. This is not unlike how HyperCard was used in the 1980s to create functional prototypes in early participatory design projects [88].

From the surface perspective of the user, it was not possible to see the difference between the Webstrates-based prototype and a regular web application. Webstrates is built within the ecology of the web, hence frameworks and libraries creating a familiar look and feel could be used (T7: Creating Something New vs. Offering Something Familiar). PARTICIPATE was developed by professional programmers that used their everyday tools by adapting our Webstrates file system integration to integrate with their own version control and issue tracker system (T8: Working With Your Own Tools vs. Adopting Built-in Tools).

With Webstrates, we have yet to figure out how to realize fine grained access control. Currently, a user logged into a Webstrates-server can have read and/or write access to a webstrate such as an activity sheet in PARTICIPATE. In PARTICIPATE, users who were logged in with an organizer account could configure an activity sheet. However, ordinary users also need write permission to the activity sheet to,
e.g., create new posts. Hence, the security model we employed was light (what we have been referring to as toilet door security) and the configuration UI for ordinary visitors was just hidden through CSS. Anyone a bit web development savvy could in principle open the developer tools of their browser and edit the activity sheet—something we never experienced. This addresses two tensions. Firstly, T1 (Malleability vs. Stability): To rapidly iterate on prototypes we leveraged a platform where all edits could be done on the client side sacrificing the stability of server side programming. Secondly, T4 (Big vs. Small Distance Between Development and Use Views): From the perspective of the system using and developing is technically indistinguishable and it is impossible for the system to guess if an edit to the DOM is considered the one or the other. This, we believe, is an inherent tension in computational media.

13.6.3 Programming Assignments

We have used Webstrates on multiple occasions to develop tools for teaching, and we will highlight two of them here. In the first occasion, Webstrates and Codestrates v1 were used to facilitate programming assignments in an interactive systems class [53]. 58 students worked in pairs developing small interactive applications, which would be developed inside the notebook interface. Multiple exercises could co-exist in the same notebook, together with instructions and notes. The workflow relied heavily on the use of the built-in package manager [52]. Teaching assistants (TAs) would prepare programming assignments in a private codestrate and push assignments as packages to an assignment repository. Students would then pull the assignments to a codestrate that they collaborated on in pairs. When they solved the assignment they handed it in by pushing it to a “hand-in codestrate.”

In another occasion, Codestrates v2 has been used to develop a tool to create interactive exercise sheets for programming (see Figure 13.8). These exercise sheets have been used for multiple years in an introductory class to programming with over 100 students. The instructor would author exercises through Cauldron with an extension for easily instantiating new exercises with various types (JavaScript, CSS, or DOM manipulation exercises). The instructor would write small unit tests for the exercises, which would automatically be run in the students’ sheet for immediate feedback. Each student would generate their own copy of the exercise sheet of the week, and hand-in the URL to their TA for grading. The TA could leave feedback for individual exercises simply by appending a parameter to the URL of a codestrate, which would show a textbox per exercise for comments.

These two uses of Webstrates and Codestrates for teaching touches upon several tensions. In the case of the exercise sheets, using them with 100 students required a certain level of stability (T1: Malleabil-
13.6 Case Studies

We therefore hid the ubiquitous “Edit” button, which was simply done through a line of CSS. Furthermore, the exercise sheets ran in protected mode to avoid browser extensions to break the document. Finally, some exercises required DOM manipulation. Here, we sandboxed the area the students manipulated in an `<iframe>` to avoid, e.g., a line such as the following that would empty the whole document:

```
document.body.innerHTML = ""
```

It is worth noting that the students always had the opportunity to edit an assignment through the developer tools or open Cauldron by appending the parameter `?edit` to the URL of their sheet. The latter would give access to the unit tests from where they in principles could derive the assignments’ answers. However, we never experienced this to be a problem.

In the interactive systems class, students generally were excited by the possibility of real-time collaboration on solving exercises. However, they also struggled with edits that would conflict in one way or the other (T2: What is Shared vs. What is Not Shared). For example, if a student reloaded the page while another student was editing a script that then would not run, or when a student collapsed or opened a code editor that would make the window of the other student scroll.

As mentioned above, in the interactive systems class assignments were distributed using the package management of Codestrates v1. This made it easier to correct mistakes in the assignment at a later point (T6: Self-contained vs. Auto-updating Authoring Environment). Still, updating an assignment would reset the code already written by students. In the programming exercises using Codestrates v2, the exercise framework was also dynamically distributed as a package. The programming assignments themselves, however, were embedded

![Figure 13.8: Codestrates v2 based exercise sheet for teaching programming. Left shows the students’ view of the exercise, right shows how the instructor can edit the exercise using Cauldron.](image)
into the prototype codestrate students copied. Here, changes to the
programming exercises would require students to create a new copy
and move over code from their old one.

13.7 The Reprogrammable Game Challenge

After we finished the development of an initial version of Codestrates v2 and Cauldron in 2020, we used the prototype in a study with programmers. The aim of the study was to get open-ended empirical data on the experience of using Webstrates and Codestrates v2. Additionally, to evaluate the usability and understandability of Webstrates and Codestrates v2 to further improve the platform in the future. To make the tasks for participants more playful and explorative, and to encourage collaboration, we decided to conduct the study as game challenge to implement a reprogrammable multiplayer game. That is, the outcome of the game challenge itself was supposed to be a piece of computational media.

The study was not conducted to evaluate the tensions. The tensions were articulated after the study was conducted. However, we use the tensions as an analytical lens to analyze the findings of the study. This is described in more detail in Section 13.7.4.

13.7.1 Participants

We recruited participants for the study creating posts promoting the study in the Slack workspace of Webstrates and Twitter, as well as inviting potential participants directly, e.g., people that used Webstrates before. Participants were required to have experience in web development and JavaScript programming — experience in game development was not a requirement, and prior experience with Webstrates was not a requirement either. 23 people signed up to participate in the study, 12 of them participated in the interviews, and 9 of those filled out the demographic questionnaire. Three interviewees described their gender

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
<th>P9</th>
<th>P10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming</td>
<td>8</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Web Development</td>
<td>6</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>JavaScript</td>
<td>8</td>
<td>2</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>Game Development</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Webstrates</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Codestrates v1</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 13.1: Overview of the participants’ self-assessed programming knowledge on a scale from 1 (no knowledge) to 10 (expert). (Only participants that filled out the demographic questionnaire are listed.)

218
as female and six as male. Interviewees’ age ranged from 24 to 37 years. Five interviewees described their occupation as being Ph.D. students, one as a postdoc, two as researchers, and one as a software engineer. Participants’ self-assessed programming knowledge is summarized in Table 13.1. Some participants worked together in groups. Table 13.2 summarizes the groups and the games they implemented.

13.7.2 Procedure

The study was conducted in June 2020 and took place entirely remotely, due to the COVID-19 pandemic and the diverse locations of participants. After recruiting participants, the study was kicked off by sending participants a design brief describing the task and scope of the study. Next, participants had three weeks of coding time to work on their games in their own time. After the coding time, participants presented and demonstrated their games in a joint virtual meeting. Finally, we conducted semi-structured interviews with 12 participants. Participants were not compensated for their time and volunteered in partaking in the study.

13.7.2.1 Design Brief

In the week before the game challenge, participants received an email with the precise schedule of the study and an invitation to a Slack workspace that we prepared for the study. At the first day of the game challenge, participants received a design brief including the task description of the study, documentation of the Webstrates, Codestrates v2, and Cauldron platforms, and a link to the consent form and a demographic questionnaire.

The challenge consisted of four tiers: (1) Make a small game, (2) make it multiplayer, (3) make the rules or part of the rules editable through programming, and (4) allow users to edit the rules collaboratively (i.e. anyone can change the rules at any given time). Participants were encouraged to fulfill all tiers but it was not a requirement. There were no restrictions in what game participants could develop and they were allowed to modify the examples we provided them with. Participants were given the opportunity to work alone or in groups with other participants. For the challenge, participants should only use Cauldron for the development of their game and

<table>
<thead>
<tr>
<th>PART.</th>
<th>G1</th>
<th>G2</th>
<th>G3</th>
<th>G4</th>
<th>G5</th>
<th>G6</th>
<th>G7</th>
<th>G8</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAME</td>
<td>uruburu</td>
<td>–</td>
<td>Card Game</td>
<td>The Teachable Game</td>
<td>Flappy Bird</td>
<td>Tank Game</td>
<td>–</td>
<td>Rouge</td>
</tr>
</tbody>
</table>

Table 13.2: Overview of the groups that worked together and the titles of the games they implemented.
no external editors. Participants were, however, allowed to use any JavaScript libraries, such as Phaser, p5.js, or Three.js.

The documentation of the platform included the documentation of Webstrates, Codestrates v2, and Cauldron. We created a “getting started” video, introductory documentations, and full API references for the three platforms. Furthermore, we created examples of small applications using Cauldron (a todo-list and a simple slide show application) and two games implemented in Cauldron (a “Baba Is You” inspired puzzle game and a “Tank Trouble” inspired multiplayer game; see Section 13.8). The examples were intended as a starting point for participants to see how applications and games could be implemented in Cauldron.

13.7.2.2 Coding Time

After receiving the design brief, participants had 16 days (later 21 days, see below) to work on their games. Participants could themselves decide when and for how long they wanted to work on their games. In the provided Slack workspace participants could communicate with and receive help from each other, as well as, receive support from the facilitators of the study and the developers of the Codestrates v2 and Cauldron platforms. Participants were also encouraged to report any bugs they encounter in the Slack channel.

After inquiring on Slack after one week of the challenge how participants were progressing with their games, we found that many had yet to start with the challenge. Thus, we decided to extend the timeframe by another five days, resulting in a total of 21 days for participants to work on the game challenge.

13.7.2.3 Game Demonstration

After the coding time, participants were invited to present and demonstrate their games in a joint virtual meeting. Participants presented five games live and one game in a recorded video presentation, as this group did not have time to participate in the meeting. After each presentation, participants had time to discuss and comment on each other’s games. The meeting lasted one hour.

13.7.2.4 Interviews

In the week following the game demonstration, we conducted semi-structured interviews with participants. We conducted ten interviews with a total of 12 participants — eight interviews with an individual participant and two interviews with two participants. Each interview was semi-structured and covered topics such as what game participants developed, how far they got, what inspired their ideas, as well as questions about using the Webstrates, Codestrates v2, and Cauldron platforms. For participants that worked in groups, we also asked
for their experiences in collaborating in Cauldron. The duration of interviews varied between participants from only around 15 minutes up to 75 minutes; most interviews lasted between 30 and 45 minutes. This depended, among other things, on the progress that participants made on their game and their prior knowledge of Webstrates.

13.7.3 Data Collection

We collected demographic data about the participants in the beginning of the game challenge. We recorded the audio and video of the game demonstration meeting. All resulting games and their code version history were preserved on our servers. We recorded only the audio of some interviews, while for others we also recorded video, which included screen recordings of the participant showing their game as well as drafts and resources they used for inspiration or help in the development process. All of these data were associated with pseudonymized IDs and stored on secure university servers. Furthermore, we collected the game prototypes that the participants implemented.

13.7.4 Analysis

We first transcribed all interviews verbatim. For parts that were difficult to hear, we either omitted that section or provided two or three likely alternatives so as to keep the transcriptions as close to the actual wordings as possible. The Danish interviews were then translated into English, striving for equivalence in meaning and interpretation in favor of transliteration [258]. This was less a problem in our context, as Danish and English are closely related, and our interviewees largely share cultural backgrounds. When reporting quotes from interviews in this paper, we show “cleaned up” versions for legibility that preserve the original meaning. For instance, instead of “I- I- I think that” we write “I think that.”

In analyzing the interviews, we followed a two-stage thematic analysis [55] approach. In the first stage, we used the eight tensions in Section 13.5 as the guiding themes for a deductive coding of all interview transcripts. We chose a deductive approach in analyzing the data as a means to reflect on how the design and engineering decisions behind Codestrates v2 shaped users’ experiences with and understanding of computational media. During this first stage, we focused on identifying participant comments that reflected any of the eight tensions and their associated concepts (e.g., comments related to malleability, stability, and the tension between them). One author coded all interviews, while three other authors each coded two or three of them, so that all interviews had been coded by at least two authors. The coding consisted of associating interview fragments with one or multiple tensions on a spreadsheet, as well as writing notes.
about how each fragment pointed to insights about user experiences in the scope of the tensions. The second stage involved a detailed discussion of all interviews, mixing deductive coding (to discuss the data in terms of the tensions) with inductive coding (to create sub-themes). Over multiple meetings, four of the authors went through all of the interviews together, line by line, and discussed why they were coded for which tension. The main goal here was not to find agreement between coders for achieving more “reliable” results, but rather to use disagreements and differences between coders as discussion drivers [212]. The discussions aimed at identifying the most salient and interesting patterns in participants’ experiences, which led us to creating subthemes as well as reflecting on our “lessons learned” about realizing computational media.

The tensions in their initial definition in Section 13.5 were derived from insights on the historical process of designing and prototyping computational media with Webstrates. In this game challenge, however, we used them in the context of participants using computational media. This made it possible for us to identify interesting experiences related to the tensions. Still, this analysis was not intended to evaluate the tensions themselves, but for us to reflect about the impact of our current design trade-offs for the Codestrates v2 platform.

Having worked with Webstrates and Codestrates to varying degrees in prior projects, we as authors might have had biases towards some tensions and reflected on our own use of the Webstrates and Codestrates platforms. We, further, also incorporated data that lied outside of the interviews in our discussions: We used both the games that participants implemented and the video of the game demonstration for clarification.

13.8 THE GAMES

Before we present the results of our study in the next section, this section will summarize the games the eight groups have worked on during the study. While most groups implemented or started implementing a game, G2 and G7 did not manage to implement a game. For these two groups, we report on how they went about the game challenge.

13.8.1 Group 1: Uruburu

P1 worked alone on his game Uruburu (see Figure 13.9a). The game of P1 is inspired by the video game Snake\textsuperscript{15}: A snake with two heads is used to move through a maze with the goal to reach one head with the other. This is done by eating items to grow. The multiplayer aspect

of the game would be that each head is controlled by a different player, thus, they have to collaborate to solve riddles.

P1 spent a considerable amount of his time ideating about the gameplay, multiplayer, and how to make the game malleable. Being familiar with Webstrates, P1 aimed to represent the levels as SVG vector files that would be synchronized in the DOM of Webstrates, allowing to be inspected and modified by users in the developer tools.

P1 got started with implementing the levels, design of the snake, and some basic gameplay mechanics like moving the snake or adding walls the snake cannot pass through. One head could be controlled using the WASD keys, the other using the arrow keys. Further gameplay mechanics, like growing, eating items, or multiple levels, however, were not yet implemented.

13.8.2  Group 2

P2 tried to use Codestrates v2 and Cauldron alone but did not succeed due to missing programming knowledge (see also Table 13.1) and time to spend on the game challenge. P2 felt overwhelmed by Cauldron and its complexity and struggled to use basic functionality like creating fragments and toggle them to auto-run. A reason was that the development environment in itself was not self-explanatory and did not support users in programming as other block-based environments P2 tried out in the past. Due to these difficulties, P2 did not attempt to create a game.

13.8.3  Group 3: Card Game

G3 originally consisted of P3 and one of the authors who acted as a participant-observer during the ideation phase and early stages of the prototyping phase. The researcher let P3 be in charge of implementation details and other technical decisions. After this, P3 worked on their own to develop and implement the game. Importantly, the author did not participate in the demo session, was not interviewed, and did not interview P3.

G3 developed a card game based on Uno that would accommodate up to four players. In contrast with the original Uno game, this game was meant to allow players to dynamically alter the rules along the way. Further, the rules were envisioned to be arbitrary and not limited to a particular set of pre-defined rules. Importantly, the game was designed as to not enforce rules immediately so that players could potentially break the rules and get away with it, unless another player explicitly asked the system to check if a rule was broken.

---

16 Uno card game: https://en.wikipedia.org/wiki/Uno_(card_game)  
(Retrieved February 22, 2022)
P3 decided early on to develop the game in their own code editor Visual Studio Code with the plan to later migrate it to Codestrates v2 and the Cauldron editor. This ended up causing issues with sharing the game among multiple clients as will be unfolded further in the findings. Therefore, the game only worked locally on a single computer at the time with all participants having to be co-located. This also meant that every player could potentially see all players’ cards.

13.8.4 Group 4: The Teachable Game

G4 consisted of P4, P11, and P12 who based their game design on two core technologies, WebRTC and Tensorflow, that P4 and P12 had
previously worked with respectively. The game was conceived as a “hot potato”-style\textsuperscript{17} game with either a timed bomb that you need to pass on to other players or a randomly moving bomb that you need to avoid.

The playing field is a Windows 95-like desktop dotted with folder and terminal symbols, and each player is represented with their live video feed in a small window that can move around based on tracking gestures. The game allows for two different models of tracking: face and hands. To move the video feed avatar, a player moves the respective body part around in the window based on a pre-trained machine learning model.

To alter the rules, a player must move to a terminal symbol, which then prompts the player for a global rule change such as reversing the $x$- or $y$-axis or changing the tracking model. The other players are not explicitly notified that any changes have happened but they immediately go into effect for all players. The game was not fully finished as the bomb element was never written. Thus, it was possible to play the final game but there were no conditions for winning or losing.

13.8.5 Group 5: Flappy Bird

P5 worked alone in creating a Flappy Bird\textsuperscript{18} clone for multiple players. Each player controls a flying bird on a continuously moving background with pipes of various lengths that they then need to avoid by flapping. The only control is the space bar that makes the bird flap upwards to counteract gravity pulling the bird down. The game field is synchronized among all players and each bird is likewise synchronized among everyone, thus making it possible to see where the opponents are flying. For making the rules editable, P5 separated some of the game logic in a single fragment that they then exposed to the player as part of the game window. In essence, the game window consists thus of both the actual game and the program code in which you can manipulate, e.g., parameters for gravity or amount of pipes. The game field is dynamically generated, and the game ends once every bird has crashed into the ground or a pipe. The player with the highest score, i.e. number of pipes passed, wins.

13.8.6 Group 6: Tank Game

P6 and P8 worked together to create a 2D tank game. Unlike the other games that came out of the challenge, the tank game is a direct adap-

\textsuperscript{17} Hot Potato game: https://en.wikipedia.org/wiki/Hot_potato (Retrieved February 22, 2022)

\textsuperscript{18} Flappy Bird: https://en.wikipedia.org/wiki/Flappy_Bird (Retrieved February 22, 2022)
tation of an example\textsuperscript{19} given to the participants by the authors. The participants decided to keep most of the original gameplay mechanics. That is, the game is distributed among clients, each player controlling a tank with the WASD keys and firing bullets with the space bar. If hit by any bullets in the game, the player is then disintegrated and can respawn to play again.

P6 and P8 implemented a portal that—by shooting it—would transfer all players to a new “world” with a different set of rules that you then would have to explore by playing. While originally considering having players edit the rules manually in the program code, the group decided that this would break the pacing of the game and thus opted for a different approach. When the portal is shot, a random parameter related to the game mechanics is changed to a random value, e.g. the tanks would rotate faster or slower, or you would have more or fewer bullets available. The portal moves around randomly on the playing field and its position is not synchronized among players, meaning that each player sees a unique position of the portal.

While the game is playable in its current state, G6 also considered implementing a scoring system, a leaderboard and other game modes such as team deathmatch but due to time constraints never did.

13.8.7 Group 7

Much like P2, P7 did not end up developing a game. In their own words, it was mainly due to lack of game development experience. The intent was to create a game in the style of The Incredible Machine\textsuperscript{20} in which the goal is to construct elaborate Rube Goldberg-esque contraptions. P7 explains how the rules that were to be altered would then be parameters of the game physics such as gravity and bounciness of particular elements. While they did manage to create an application through Codestrates v2 and Cauldron containing a few game elements, P7 struggled to implement a working physics model and thus ended the challenge rather early.

13.8.8 Group 8: Rouge

The single-player 2D game by P9 and P10, Rouge, is an homage to the early dungeon crawler Rogue using emoji symbols instead of textual glyphs, each taking up a $1 \times 1$ space in the game field. The player controls a red lipstick that they can move around using the arrow keys. The purpose of the game is to delve deep into the dungeon. On every level is a set of stairs through which the player will be

\textsuperscript{19} Codestrates Tank Trouble example: https://demo.webstrates.net/TankCauldron/release/?copy (Retrieved February 22, 2022)

transported to the next level, and each level in Rouge consists of a number of automatically generated rooms and hallways that the player must explore.

One salient feature of Rouge is the commands. By using the letter keys on the keyboard, a player is able to write on the ground in a given direction so as to form commands that enforces rule changes, e.g., giving the player the ability to walk through walls or becoming solid again. However, as the player moves along with the writing and the commands have to fit inside a particular room, finding the right spot to write commands becomes a core part of the game strategy. This was motivated by the game Baba Is You. G8 further planned for providing multiple lipstick colors, each enabling a particular writing mode. Ultimately, however, the game was not fully finished, being judged by P9 and P10 themselves to only be around halfway done.

13.9 RESULTS

We analyze our findings through the lens of the tensions introduced in Section 13.5. We report on six out of the eight tensions. The tensions T5 (Authoring Environment Written in Itself vs. Not Written in Itself) and T6 (Self-contained vs. Auto-updating Authoring Environment) did not reveal new insights. For each of the six tensions, we present relevant subthemes that turned out to be of particular interest in the interview data. These subthemes are not meant to be exhaustive of each tension; rather, they emphasize aspects that were surprising, interesting, or otherwise illustrate open challenges for future work on computational media. Each section is concluded with a short summary of takeaways from the findings.

13.9.1 T1: Malleability vs. Stability

The tension between malleability and stability affected most participants. It manifested both in the versioning system of Webstrates, which enables to roll back changes, as well as in defining explicit safeguards to the code by defining hot and cold spots in the code.

13.9.1.1 Versioning

Various versioning practices turned out to be central safeguards to accidents. While versioning is core to how Webstrates operates and Webstrates provides a versioning API with an integrated revision browser in Cauldron, not all perceived it as a real versioning system that could be trusted. Some explained this was due to feeling a lack

Baba Is You: https://en.wikipedia.org/wiki/Baba_Is_You
(Retrieved February 22, 2022)
of control in what changes comprised a version and that is was “not transparent” (P10).

You don’t really know how many steps to go back. Because, is it a line of code added, is it a character, is it like the last half-hour that I’ve been working? (P9)

Indeed, code fragments are elements in a webstrate’s HTML, thus every change to the code—including the addition of a single character—is a change to the DOM that is persisted in a new version. However, participants seemed to disregard that Cauldron was just a window to editing the DOM and expected a more traditional way of versioning “just the code” of their game. For example, P3 was aware of the revisions browser, but was expecting it to focus on the changes to the code fragments and not the malleable system as a whole:

When you navigate the revisions, it shows you the website [i.e., the game]. And sometimes there were changes in the code that weren’t reflected. So it was difficult to know where were the differences in the code base. (P3)

To have a better sense of control over the code versions, G8 ended up using Google Docs as a make-shift versioned archive. P9 considered Google Docs a “safe place” to archive their code in. Others used the prototyping capabilities of Webstrates to recreate a working version when something broke. The tank game group had started their project from one of the provided examples, so when their application broke, they created a new copy of the example and copy-pasted their code piecemeal to identify the bug and recover from there.

An important consideration in designing future tools might be to allow users to separate versions caused by changes in program code from versions caused by application use. As Webstrates has automatic built-in versioning, the development and use activities are conflated with regards to versioning. This conflation of versions, in turn, rendered the versioning system useless as illustrated by G4:

We took the kind of bad choice that we also saved where every player was, we also saved their position in a JSON fragment. And that had the effect that every time a player moved, a new version appeared. [. . .] We were up in like 30,000 or something. And that actually made it kind of useless to roll back. (P4)

All participants developed working strategies to cope with breakdowns, and none experienced significant loss of work due to the malleable nature of the system. Interestingly, some of the wishes of participants, e.g., HTML fragments using a persisted DOM (see Section 13.9.2) would have exacerbated some of these issues even further, as in these cases even simple changes to the state of the game would have created new versions in the version history, convoluting it even further.
13.9.1.2 Hotspots vs. Frozen Spots

Participants made explicit efforts and code-structuring decisions to control what parts of the games players should or should not change. For instance, P5 used the capability of Codestrates v2 to instantiate an editor in the view for a specific code fragment to expose a part of the game logic as reprogrammable, e.g., the “gravity parameters” (P5). P1 intended to make their game live editable from within the game itself, without having to switch to a separate level editor:

“Yaga is you” is a similar sort of tile based puzzle game and it has a level editor, but the level editor is is an alternate page that you access by putting in an additional argument in the URL. And I wanted it to be more live editable. So my ambition was […] actually be some invisible elements that you can make visible by pressing a button, to open the editor, that would switch the game into editing mode. (P1)

We observed a pattern where parts of the game would — in framework terminology — be considered a hot spot while the rest were frozen spots to the players. Hot spots usually had a dedicated fragment, separating them from the rest of the code. The hot spots created by the participants included choosing between different rule sets through a drop down (in G4’s Tensorflow-based game) and directly editing parts of the game code as in P5’s Flappy Bird clone.

P5’s game was the only game where code was directly exposed for editing within the game interface (i.e., without having to open Cauldron) and where the player could break the game by, e.g., introducing a syntax error. P5 imagined creating safeguards for these by using block-based programming for the hotspots to “make it simpler for non-programmers to like sort of understand what’s going on.”

Other participants like P7 also voiced their concerns about granting players uncontrolled access to the whole source code. For instance, P7 voiced that “if they can open the editor and change the way things bounce around, well they can change anything.”
13.9.1.3 Takeaway T1: Malleability vs. Stability

Changes by users to malleable software, unsurprisingly, not always work out as planned. This can inflict stability. To ensure confidence in users changing malleable software they need to be able to trust the versioning system, being able to roll-back changes. While Webstrates provides extensive versioning, it is ill understood by many users rendering it useless for them. Versioning needs to be meaningful, even more when the versioning of both the code and the application state share the same space. When it comes to making applications malleable and reprogrammable by other users, we found that participants were aware of these issues and considered exposing only certain hotspots of the code to users while keeping the rest as frozen spots. Here, malleable software needs to provide more mechanisms to support programmers in creating these safeguards.

13.9.2 T2: What is Shared vs. What is Not Shared

Participants faced many confusions and surprises as they found differences between what they expected to be automatically shared (i.e., synchronized across clients) and what Codestrates v2 actually shared. At the same time, some parts of documents were automatically shared but participants expected them not to be shared.

13.9.2.1 Expecting Fragments to Share Their DOM Output Across Clients

Some participants assumed that because fragments are shared (i.e., collaboratively editable), their output is shared as well. This association is often implied in the way these participants talked about fragments, e.g., “those sort of fragments, they are not synchronized between clients” (P4) — fragments are synchronized between clients; their DOM output is not.

Participants that had previous experience with Codestrates v1 and Webstrates were confused when noticing that the DOM output of the fragments was not automatically shared. While this decision was made to better support collaborative editing of a fragment, participants that collaborated in Cauldron still expected the view of fragments to synchronize as in Codestrates v1. For example, P1 carefully designed their game entirely based on DOM elements to automatically synchronize all players: “I also felt lightly betrayed by the tools because I felt like the very fact that you didn’t have to do anything to persist [share] stuff was the key value of Webstrates.” P4 had previous experience on Codestrates v1 and had similar expectations.
It’s as if some of the idea that everything just synchronizes, that’s removed a bit now, that it had been drawn back. And that can be a little sort of counter-intuitive because I think, like, well my basic thought is “okay, I have this DOM that just synchronizes.” And then all of a sudden, then, then it, it doesn’t actually do that. (P4)

13.9.2.2 Expecting Fragments to Share State Across Clients

A common surprise among participants was to realize that the automatic sharing of a webstate’s DOM did not come with automatic sharing of state, e.g., instantiated JavaScript objects. In other cases, some used JSON fragments to parametrize game rules and system properties (e.g., what actions or penalties apply to a player when they draw a card from a deck) and faced similar surprises when noticing that editing a JSON fragment did not update the state of the system but required a page refresh.

P3 designed a Uno-like card game where the cards that are drawn from a deck hold the rules about what should happen next, e.g., “the two of hearts makes the next player draw two cards.” The deck of cards should be shared among all players, and each player should see their own hand but not the one of other players. Rather than developing the game with Codestrates v2 and Cauldron from the start, P3 chose to first build a prototype with Vue.js, a JavaScript framework that keeps views synchronized with their models, and then move the code to Codestrates v2 to make the game multiplayer.

P3 expected the views of common game elements such as the deck and discard pile to automatically synchronize when their Vue.js models changed in one client (e.g., when a player draws a card). However, once P3 started testing the game with multiple clients they realized that the state of the deck and the cards in players’ hands were not shared automatically: “So we wanted all the players to have the same shuffled deck and that’s not the model of Codestrates. You don’t share state, yes, you only share view.” As a workaround, P3 tried to use a JSON fragment as a shared model across players, expecting it to work as a live object with synchronized state across clients, which it, however, does not: it only synchronizes the DOM content between clients.

G4 used a workaround to use a JSON fragment as a centralized model to share state across clients. In their case, each player had a camera feed on the gameboard, and their head movements controlled the position of their feeds. The JSON fragment was updated by each client with the coordinates of the camera feeds, and all clients listened for changes to this fragment to reload the JSON and update the position of the players on the screen. However, it became clear that JSON fragments were not designed to support the requirements of such

---

22 Vue.js: https://vuejs.org/ (Retrieved February 22, 2022)
real-time interactions and P12 reported that in hindsight they should have used signaling to broadcast player positions instead of persisting them in the JSON fragment.

We find it interesting that these participants were aware of the signaling mechanism recommended to communicate state across clients, but still tried to find other ways of having their state automatically shared. There seems to be an expectation that Webstrates magically takes over all management of collaboration and sharing. This is, however, not the case as merely the DOM is synchronized by Webstrates, which in turn creates confusion or disappointment among users. Section 13.9.5 elaborates on this confusion about how things were meant to be done.

13.9.2.3 Expecting Canvas Elements to be Shared Across Clients

Some participants based their multiplayer games on running examples of codestrates (e.g., the tank game) or open source, single-player games. In many cases, these games used the HTML `<canvas>` tag for rendering graphics, which lacks a DOM representation and, thus, cannot be shared across clients with Webstrates. This felt discouraging to those counting on Codestrates v2 to take care of synchronizing the UI across players. P7 realized that the tank game example did handle synchronization of the canvas-based game using signaling and felt that this would be “a bit scary” to do manually.

P8’s group G6 worked on modifying the multiplayer tank game offered in the Codestrates v2 documentation by adding a “portal” that changed the rules of the game every time a player shot at it. This portal moved around the game, so that players shooting at each others’ tanks could accidentally hit it. The portal was also part of the `<canvas>` element of the game, and when they noticed that its position was not synchronized across clients, they decided to keep that as a “fun” element in the game that adds “additional randomness” (P8).

Interestingly, even participants that used signaling to synchronize the parts of a `<canvas>` across clients sometimes took syncing for granted. P5 adapted a Flappy Bird game to be multiplayer, where two birds had to jump up and down together while avoiding the same green pipes. They used signaling to synchronize the position of the birds, but when testing they noticed that the game still looked different across clients because the location of pipes were not synchronized.

The vision of computational media lead participants to think that Codestrates v2 and Cauldron are “magical” platforms that solve most issues in programming. Some participants, for example, developed an ideal “fantasy” of the system solving all problems related to the synchronization of state for them. G4, for instance, handed the responsibility of changing their state to the platform:
We probably thought, like, that: “Ah, it’s super easy, this part about synchronizing state, because it’s practically solved for us.” And then we didn’t really spend time on writing that part of the code properly. And it, then that turned out to bite us in the ass pretty hard. (P4)

P7 was similarly attempting to implement a game with a library that puts the game into a <canvas> element, whose content was not synchronized in them DOM: “I was like: ‘But we have Webstrates.’ But I get it, it’s in a canvas.” Even though being aware of the fact that a canvas is not synchronized by Webstrates, P7 still expected Webstrates to solve this issue. P4, further, described a feeling of Webstrates simplifying everything about implementing games:

Then you can really have this idea or get sort of this feeling that “oh, then you can just sit and do all kinds of things and do things and build things super fast and such.” But when things are, like, asynchronous and it’s an event system and stuff, then you actually still have to think quite a lot if you would like to end up with something that works in the end. (P4)

13.9.2.4 Running Unfinished Code

Unlike text produced in the collaborative writing of documents, e.g., as in Google Docs, the code typed on Cauldron needs to be syntactically correct code at the time of execution. Live sharing of fragment changes across clients enables collaborative coding on Cauldron, however, the same code that is being edited by one user might be run by another at the same time, which may cause the system to break until the editing is complete and without errors. For instance, for P11 sometimes code would “crash completely” due to their teammate changing code in the same document. Similarly, P10 reported that syntax errors would often occur when working with a group member at the same time in the same fragment.

To avoid breaking each other’s code, G4 often ended up splitting tasks and working on “private” copies of their game webstrate and merging changes into their main copy once they got their part working.
13.9.2.5 Takeaway T2: What is Shared vs. What is Not Shared

In our study we learned that there are many nuances to what is shared and what is not in shareable and malleable software. By introducing the general notion of a “shared system,” Webstrates sparked the idea that not only everything is shared, but also that the difficulties that come with it are resolved as well. However, synchronizing the DOM of a website does not resolve issues around shared state, shared <canvas> elements that are not reflected in the DOM, or invalid code that is just being edited. Users of such systems need to be made aware of these nuances to disillusion them.

13.9.3 T3: Editing Directly vs. Editing Indirectly

Codestrates v2 documents can be edited at different levels of directness. In the interviews, we identified two strands of editing: editing the view of an application and editing its interactive behavior. Both strands show issues with the current programming model of Webstrates and Codestrates v2: the JavaScript engine and the DOM.

13.9.3.1 DOM vs. HTML Editing

Editing the view of an application can happen at two levels in Codestrates v2: (1) editing the DOM of the view directly, e.g., using the inspector of the DevTools, or (2) editing the DOM indirectly through the HTML fragment. Some participants were aware of this difference but expected them to be the same, e.g., P3 mentioned that “one thing is the DOM and the other thing is the HTML fragment, and these two entities are different, it’s not the same.”

Being able to edit elements in the DOM directly motivated P1 to store the view and its state directly in the DOM and not in a canvas element. P1 was doing this by using SVG elements that could be stored in the DOM: In P1’s game the level was created with SVG elements with specific classes like a “tile” or a “wall.” Changing the class in the browser inspector would not only change the appearance of an element but also how it behaves in the game.

Having worked with Webstrates before, P1 thought that this would also ensure persisting the view across clients. In Codestrates v2, however, HTML fragments persist the view as plain text in a single DOM element with the server and render the views only in transient elements in the DOM of each client. While this was introduced to allow users to collaboratively edit the HTML format of a view in real-time, in this case, it also caused the DOM inspector to only being able to modify the live but not persisted view, resulting in a unsynchronized
state of the view. P₄, for instance, was confused by this change as it took away the premise of everything being synchronized:

Okay, the DOM is just being updated directly, so let’s just take advantage of that. But that, suddenly it’s not like that anymore, now I need to actively update the DOM myself. (P₄)

Overall, a lot of issues of participants with editing the view of their application were related to their motivation and expectation to use the live view of their application as a way to persist state and enable multiplayer games—the persisted DOM was perceived as a “key value of Webstrates.”

While participants expressed disappointment about this change of the view not being synchronized back into the fragment, they at the same time took collaborative editing of subsubsections for granted. There is a trade-off in between supporting collaboration and trying to keep the editing as direct as possible: In Codestrates v2 we opted for supporting collaboration to create a coherent editing experience of fragments. We did this, however, at the cost of causing breakdowns when it came to the assumption of the view and contents of a fragment being the same.

An underlying problem of this might be the use of the DOM in the Web: changing the DOM in a direct way is difficult and often requires compromises, e.g., the developer tools display the DOM as HTML, yet, elements also can contain JavaScript objects that are not immediately visible in the HTML. There is a need for more advanced tools to edit the DOM live and collaboratively that does not simply serialize the DOM, modify the HTML, and deserialize it again into a new DOM—leaving old JavaScript state and event listeners behind.

13.9.3.2 Live vs. Regular Programming

Similar to the view, also the interactive behavior exists in a textual code representation and in a running and in-memory state: (1) The JavaScript code in fragments and (2) the running version of the code in memory. Editing the code representation does not change the running format in itself, it first needs to be executed or “run” in Cauldron. This, however, does not necessarily change instantiated “live” code, such as event listeners in the document, but merely adds new behavior to the document, e.g., new event listeners. Multiple participants experienced this behavior first-hand, running into various problems. P₇, for example, had the problem that re-running code only added another playfield to their game instead of updating the old one and instead needed to refresh the whole webstrate to apply changes. P₁₀ had a similar problem as P₇, where re-running code added only an additional “window”:

I think it was really cool that thing, that there was that auto-update when you were create a website. That so, when I wrote
something, that it like was just, well that you could see it in the window right away. But then we [...] couldn't get that auto-update to work. And when we then pressed "run," it just like spawned a new window down, further down in the game. (P10)

With HTML and CSS fragments being updated live in the view of an application, Cauldron also mislead participants in thinking the JavaScript would be updated in the same “live” way. P1 emphasized enjoying the quality of “live editing” CSS and HTML. For instance, P1 considered being able to change colors in CSS with a color picker and changes happening live and in a persisted way as a “really enjoyable little interaction.” P1, however, also mentioned that JavaScript did not provide this quality.

Because Codestrates v2 and Cauldron presented themselves as an environment for live editing of malleable software, participants built expectations of the system providing mechanisms to support this way of programming. Like its underlying platform Webstrates, however, Codestrates v2 also uses JavaScript and its runtime as its primary programming model. As an imperative programming model, JavaScript is not designed to be used “live” and to be “re-run” to change interactive behavior while an application is already running. This mismatch caused several problems among participants, P5, for example, struggled with trying to change the gravity value of the bird in a Flappy Bird game, where re-running the code would only update the “bird” class but not its instances.

P12 describes similar breakdowns, where “some code running that is too old [...] can accidentally hang around and ruin the other [new code].” P12 elaborates that re-running code only runs the new code on top of the old one and wished for ways to “unrun” already executed code in order to avoid having to reload the whole website in order to clean up old live code:

It would be awesome if you could just say "run" and then say "unrun" and then make an edit and then "run." So you again don’t need this refresh, because it’s something that takes, takes quite a lot of time, and you are thrown a little out of context and such. [...] I mean, it’s this thing of, when you say “run,” you like, execute a script. And then it’s sort of executed now, and if you then press “run” again, well then it runs the same code on top of it, and that messes it up. (P12)

P1, being experienced with implementing in Webstrates, explained this issue in detail and was wishing for a “more declarative language” to be used in the kind of persisted computational medium in which Cauldron is used in:

Whenever I change the JavaScript fragments I have to refresh to actually have anything happen. Cauldron theoretically allows
me to re-run the code. But, in interactive code like this, which depends on mechanisms like event listeners, that generally puts the game in a really strange state because it just keeps the existing event listeners [...] and just adds the new ones on top. Because this is procedural code, it is imperative. It says do some stuff now, rather than these are the rules. So I definitely found myself wishing that I was working in a more declarative language or framework. (P1)

The problems described by the participants is caused by the divergence [21] between the representation of software when it is running, the JavaScript runtime, and the representation of the form in which it is being edited, the code in the JavaScript fragment. In Codestrates v2 these two representation might not be the same after editing only the code without executing it again and overwriting the old code in the runtime. Basman et al. [21] discuss the issue of event listeners being “a primary source of divergence” and how this could be prevented by using a declarative registration of event listeners.

13.9.3.3 Takeaway T3: Editing Directly vs. Editing Indirectly

Overall, the study demonstrated to us that the conventional imperative programming model of web development does not fit computational media like Webstrates. We found that code that evolves over time and at runtime requires a different programming model to support the “live” reprogramming their user interfaces like Cauldron might suggest to the users. In relation to editing the DOM, we found that changing how the rendering and synchronization of user interfaces is handled, can cause a series of breakdowns as it shakes up fundamental values and assumptions about Webstrates.

13.9.4 T4: Big vs. Small Distance Between Development and Use Views

In comparison to the previous subsubsection on levels of directness in editing, this subsection concerns the conceptual and practical distance between the activity of developing and the activity of using the developed artifact. While Cauldron has a rather short distance and aimed to support faster context switches between development and use, we found that this short distance can also cause problems in relation to shared state, as well as when being paired with an unfitting programming model like JavaScript.
13.9.4.1 The Good and Ill of a Short Distance

To open Cauldron in Codestrates v2 a single click on the “Edit” button is sufficient. Once Cauldron is loaded, it is opened on the screen inside the web browser, right besides the application. Being able to switch from using a game to developing it in such a small distance was perceived as a benefit by participants. P₄ liked the fact that one can “just go in and see ‘okay, how is is implemented?’” Also P₁₂ liked this behavior:

But if there then is something that suddenly doesn’t work, you can go in and inspect what it is, you can just jump to that one and things like that, and you have your development environment in the same place. It works well for this kind of prototyping-in-the-wild-ish thing. (P₁₂)

Being right besides the application in the same browser window, however, also caused problems when participants mixed up the current context they were working in: This led some participants to accidentally write gibberish in the open fragment in Cauldron while they thought they were interacting with the game. P₈ provided an example of the game and the Cauldron both “fighting for” keystrokes:

So we had this WASD mapping and that means when you press that [key] your tank moves around. But that also meant that if you didn’t clock out of Cauldron, it would write into the text file [fragment] that you were working on, the W-A-A-A-S-D and so on. And so that causes a bit of trouble sometimes when you went to debugging and testing if it worked what you implemented, but you will, at the same time be messing up your file. (P₈)

13.9.4.2 Shared State Between Development and Use Environment

We also learned that the shared state between Cauldron and the game or application can be problematic: As the memory and processing resources are shared between the development environment and the game, they can affect each other. Especially the game affecting the development environment is a rather uncommon case in conventional programming practices. P₄, who was working with a computationally heavy application, explains this well:

As soon as it had run, and even if it had stopped again, then I had to refresh the whole browser, because it couldn’t write anything at all. So much had it slowed things down. And you couldn’t write anything while the application was running. I mean, you couldn’t be working with WebRTC and have those machine learning models running, and write at the same time. (P₄)
Another problem of shared state between the development and use environment was that it was possible to not only break the code of the application being developed, but also to break the editing environment itself. P7, for example, introduced a bug that would prevent the page to load at all: “How can I even fix this error if I cannot see the code.” P3 had a similar problem where their game was covering the “Edit” button so that they could not access Cauldron anymore.

13.9.4.3 Effort in Context Switching

Since Cauldron is part of the application space, it is closed upon refreshing the browser, causing the developer to lose their current development context, i.e. scrolling positions in the code. To avoid this, P12 opened Cauldron in a separate window to preserve the development context while being able to refresh the game in another tab. However, not all participants realized this:

So usually when I’m working on the web, I have a live environment where whatever changes I make to the files, automatically propagated to the website and refreshes. So the experience [with Cauldron] was quite similar. However, when I worked with Cauldron, this meant that whenever I refresh the page, I would have to open the IDE again and go to the file that I was working at, because it wasn’t usually opening that file that I was working at. (P8)

P1 specifically mentioned their frustrations with the time spent pressing the “Edit” button and waiting “a few seconds for the Cauldron editor to appear” every time the website was refreshed.

13.9.4.4 Takeaway T4: Big vs. Small Distance Between Development and Use Views

Keeping the distance between development and use of malleable software short can cause breakdowns for users when they mix up the current context of the system and can, for instance, cause unwanted changes in the development view while testing an application. Another issue of our participants was that the development view was not “protected” and could be affected by application code, which might either occlude the development environment or even destroy it. Lastly, due to the short distance the development view was in the same memory of a webstrate than the application view. Thus, reloading the application by refreshing the website also caused Cauldron to be reloaded.
13.9.5  T7: Creating Something New vs. Offering Something Familiar

Our participants had varying degrees of experience with programming and with Webstrates. Codestrates v2 and Cauldron aimed to offer participants something familiar: a code editor like experience for programming. However, not all details matched up with the participants’ prior knowledge and created tensions in their understanding. Participants, for example, had difficulties matching the concept of a fragment with their existing concept of a file. Similarly, they struggled to understand how the synchronization of content should be facilitated using fragments.

13.9.5.1  Conceptual Blend Between Files and Fragments

There was a clash between traditional web programming using files and the way Codestrates v2 and Cauldron handled the execution of fragments: Many groups compared their understanding of fragments with the one of files in a folder system, where each fragment corresponds to a separate file. While this conceptual model worked well for HTML and CSS fragments as they are stateless, participants’ conceptual blend broke down with JavaScript fragments, as those could be both “run” and “auto-run” (see also Section 13.9.3). Participants struggled to match this execution model with their experience of executing JavaScript files in regular web development. P10, for example, stated that they “couldn’t get that auto-update to work” while referring to the auto-run functionality of fragments. It seems that P10 thought that auto-run would directly update their game while they change the code instead of just running the code on page load.

P8, on the other hand, who had experience with computational notebooks like Jupyter, could match their understanding of executing fragments individually with the one of Cauldron. P8 also mentioned that the auto-run “concept itself is pretty easy to understand.”

We found another example of a conceptual clash in the interview with P5, who did not initially understand that the order of fragments in the tree browser was in fact the same order as they are located in the DOM, hence, also determining their execution order. This stood in contrast with conventional code editors, where the order of files in folders does not matter:

After a while I realized, for example, in this window [Cauldron’s tree browser], the sequence of all these fragments corresponds to the sequence actually in the DOM tree. [...] So, for example, all the codes [fragments] get run if you let it auto-run or something like that, then the sequence corresponds to how you put these things in the DOM so it’s, it’s run like this as we know. But then, yeah, sometimes I built another thing later so I would just
sort of arrange the sequence of these pieces to make sure it runs
what I want. (P5)

These examples illustrate how small conceptual differences can
cause breakdowns by instilling assumptions and expectations of how
the system works into the participants. Developers of systems like ours
need to be aware of these nuances and avoid setting users up for wrong
expectations that users create based on concepts familiar to them.

13.9.5.2 Confusion About How To Synchronize State Correctly

Depending on participants’ prior knowledge of Webstrates, Codestrates v1 or web development in general, they expected Codestrates v2
to work differently. These expectations shaped how they approached
the implementation of the game and their use of Cauldron, which
sometimes diverged from what Codestrates v2 and Cauldron actually
supported. Typically, the DOM would be used for the synchronization
of state in Webstrates. In examples that were provided to the
participants in the instructional material, one example, a todo list,
used a JSON fragment to synchronize state, while another example,
the tank game, used the signaling API of Webstrates to synchronize
state. Sometimes participants did not know about all these options
and assumed they had to do it in a certain way. For instance, G4 used
JSON fragments to synchronize their state as this was how it was done
in the example. They, however, quickly ran into the problem of not
being sure how this was supposed to work for their game:

Then we would probably just have run, like a direct notification
to everyone about what had been changed. I mean, now it kind of
just gets this “something has been changed,” and then everyone
kind of builds and look “okay, how does this shared state that we
have, well, lying around.” And that gave us, well, it gave us a
bit of headache back and forth, you really had to figure out “okay,
how is it actually meant to be done in this.” (P4)

While synchronizing the state via JSON was a viable option for the
todo list example that we provided, as a todo list is quite unlikely to
have concurrent edits, it was not suitable for the real-time changes in
G4’s collaborative game. Also P12 stated: “something that we struggled
with a lot, it was the fact that these write conflicts happened down in our JSON
ting.” For this kind of scenario, where transient state of the location
of players is shared in real-time, using signaling would have been
the better solution. This, however, was not explained clear enough in
the examples we provided: Our example that used the JSON fragment
for synchronization tried to offer a familiar way of programming
collaboration through a model. In practice, however, this confused
participants as they expected the synchronization to happen directly
through the DOM elements, causing several breakdowns. P1, while
talking about how fragments would be required in Codestrates v2, even felt that these types of synchronizing state were diverging from the original vision of Webstrates, that P1 was familiar with:

And the way that it was folded into the newer API, yeah, overall this gave me a sort of a sense that this set of tools, these Codestrates [v2] and [...] Cauldron, that they were designs with the standards of programmers in mind, rather than the vision of Webstrates. [...] And at that point I felt a big discrepancy between my assumptions and the assumptions of the people who've built the tool. (P1)

13.9.5.3 Takeaway T7: Creating Something New vs. Fulfilling Assumptions

Trying to create something new while offering some familiarity to existing knowledge or systems is a balancing act: By portraiting fragments as files and DOM elements as folders in the tree browser, participants build assumptions about how these work. While this worked for some aspects, the auto-run feature caused confusion as it did not match with the concept of a file. Similarly, trying to offer a familiar way to synchronize state between clients can accidentally deceived participants into thinking this is how synchronization should happen. Careful explanation of such examples might help to prevent fundamental misunderstandings of a system.

13.9.6 T8: Working With Your Own Tools vs. Adopting Built-in Tools

This subsection relates to the use of ones own tools for development, e.g., code editors or IDEs, compared to using built-in tools that are integrated within a development platform, e.g., Cauldron. We found that participants had mixed feelings about using the integrated authoring environment Cauldron, not wanting to abandon their personalized tools and known workflows for developing software, but seeing the benefit of launching Cauldron with a single click without any setup.

13.9.6.1 Mixed Feelings About the Integrated Authoring Environment

As already addressed in the previous section on distance between development and use, there seems to be conflicting opinions on the authoring environment being conflated with the application space: P7, for instance, mentioned using Webstrates File Sync in the past to develop for Webstrates, which however sometimes had “weird behaviors” or crashed. An integrated tool like Cauldron would “avoid sort of those
problems” which would be “very good.” P8, notably, seemed to be on the fence about preferring their own IDE or using Cauldron:

What I always liked about Webstrates is that it’s so easy to have multiple people see the same content. And so I guess whenever I have to do something on the web that’s collaborative, I would just use Webstrates for it. And then one benefit I can see with having this Cauldron interface now is that I don’t have to bring my own IDE. So even though I guess I would still use my own IDE, it’s nice to have the option to just change something on the fly or to, so I guess whenever I have to do something collaborative where multiple people work on the same content on a website, then I would use Webstrates. And since Cauldron it is so easy to use, I probably would also use Cauldron. (P8)

P8 makes an explicit preference for their own tools but acknowledges the benefits of using Cauldron to such a degree that it seems likely they would use it. P1 is — much like P8 — on the fence about what potential benefits and drawbacks might mean for them: While P1 liked some aspects of Cauldron like asset uploading, they did not think Cauldron would help them “speeding up the flow of modifying things” and could even remove their “ability to use the debugging practices [they] learned as a semi-professional programmer.” The latter point highlights another big issue of custom integrated tools such as Cauldron: prior technical knowledge might not be able to be carried over in using a new tool. Likewise, P3 was less convinced by the benefits of adopting new tools:

I can open an HTML and write the script there. Okay, I can do that in Cauldron but I can do it in a text editor like. And I need to import this CSS library. Is, is helping me in any way, Codestrates to doing that? No, actually it might be a little more difficult than the document I can see online, because they are explaining how to do it in traditional way. Just like I need to add an extra step for something that I already know how to. And, and then, that’s the reason for me, like, I started it in an editor and then I felt, now I need to do collaboration and I know this would be a pain in the ass to do it traditional. I need to add WebSockets or I need to add some kind of communication mechanism. Oh, Codestrates gives me that for free, then I do a transition. So I’m always motivated by needs and there was no need for me to move, until I reached that collaboration point. (P3)

P3 is very clear about the value that each approach might give. In this sense, P3 is more pragmatic than idealistic in their choices: if the benefits of using one’s own tools seem greater than using the ones provided, then they will do that until the ratio of relative benefits turns towards Cauldron.
13.9.6.2 Challenges in Using One’s Own Devices

For Webstrates, the Webstrate File System utility allows for mounting script and style elements from the DOM as separate JavaScript or CSS files in an IDE like Visual Studio Code, allowing to make use of the IDE tools like syntax highlighting and autocompletion in the editor. However, with each subsequent evolution of Codestrates, this procedure is being made less possible as JavaScript code is not stored in script elements anymore but rather as plain text so that Codestrates itself can manage its execution.

That means that it is currently not practically feasible to bring your own tools for working on Codestrates v2. This tension is thus just as much a reflection of what participants might want in the future—or remember from working in Webstrates—as it is a conceptual discussion about tools and tool appropriation. This issue was also echoed by P1:

*I think this is a very deep issue of modifiable tools, which is, do they contain the tools for their own modification or do they allow you to bring your own? And I get the feeling, at least for me personally, with my own experience on preferences that Cauldron brings a bad trade-off there.* (P1)

13.9.6.3 Takeaway T8: Working With Your Own Tools vs. Adopting Built-in Tools

Integrating an authoring environment like Cauldron into a development platform such as Codestrates v2 enables users to directly author content with the requirement for additional tools, which especially useful when only small edits need to be done. For major development work, however, experienced programmers like most of our participants only rarely wanted to abandon their familiar tools. A system should strive to do both: offering an integrated authoring environment but allowing users to continue using their own tools.

13.10 Lessons Learned

Throughout our efforts to study the vision of computational media through developing a concrete software platform we have learned several lessons. While the takeaways from the previous section are connected to single tensions, the lessons of this section may relate to multiple tensions and are less specific to the Webstrates platform.

Some of these lessons learned may seem obvious in hindsight, and they echo previous findings in the HCI and CSCW literature. However, we deem it valuable to highlight what we learned through our experi-
ences that stood out in this particular case and had an impact on our current and future research.

13.10 Lessons Learned

13.10.1 L1: Reprogrammability Is Not the Same as Malleability

Our experiences with Webstrates have made it clear to us that making software malleable to a user goes beyond merely making it reprogrammable. This is a point Tchernavskij et al. [288] makes very clear, and our experiences confirm this. Providing users with access to the code of applications in the developer tools in Webstrates, the code paragraphs in Codestrates v1, or code fragments in Codestrates v2 is a step towards malleable software, however, it is not enough. Malleability happens in the relationship between the users’ capabilities, motivation, and trust in the software — and the technical flexibility for the software to be adapted and changed. Malleability of software is a non-trivial socio-technical concern and not merely a technical problem.

13.10.2 L2: Support for Switching Between Synchronous and Asynchronous Collaboration is Essential

When (re)programming software collaboratively, mechanisms to support switching between synchronous and asynchronous collaboration are essential. While real-time collaboration on code can be a blessing when working together, it can also be a curse if code breaks, for example, when unfinished code is run by other users, causing the software to crash. Similarly, writing code is often an iterative process and bugs in initial versions of code should not be immediately applied for all collaborators — this is why other version control systems like Git enable to commit changes once they are finished or even on different branches.

Synchronizing code changes immediately across all clients inhibits this way of asynchronous collaboration and forced participants in our study to copy code into external editors or to create copies of their webstrates to not disrupt their collaborators. Future collaborative computational media will require mechanisms for fluidly transitioning between synchronous and asynchronous editing. Similar needs have been observed in collaborative writing [176], hence, in the spirit of computational media, the mechanisms ideally should be similar for editing code and editing content.

13.10.3 L3: Values of New Concepts Needs to Be Clear to Users

When introducing new concepts for seemingly good reasons, they demand immediate and perceptible value for end-users to appropriate and use them. In Codestrates v2 we introduced the concept of a code fragment. We called it a fragment as they would usually encompass a
part of an application. We decided not call them *files* because, while—like files—they contain code, they have different properties, such as being able to be run independently and they live in the DOM rather than in a file system. We also decided not to call them *scripts* to avoid confusion with the script tag, even though they share many of the same characteristics. The value of this concept was not immediately clear to our participants and instead created confusion. Here, we underestimated the need for clear communication of the semantics of new concepts to the users. However, *how* this should be properly done, we have yet to master.

13.10.4  **L4: Collaboration in Development and Use Requires a Fitting Code Execution Model**

A medium that aims to be malleable and allows for real-time collaboration in use and development requires a fitting code execution model. Changing the interactive behavior of applications after they are instantiated is not a common case in regular programming with JavaScript. Merely providing access to the code and allowing users to edit and re-run code leaves a variety of open issues. For example, the code that is visible in an editor might not be the same version of the code running in memory, or one user might run a different version of the code.

The conventional JavaScript execution model is simply insufficient for such a task. Instead, the code execution model should fit the vision—in our case computational media—and provide mechanisms to change the interactive behavior of applications live, both locally and across multiple clients or users. This insight motivated our work on a new declarative programming model, *Varv*, suited for computational media [51], that, we believe, represents the next step in our exploration of computational media.

13.10.5  **L5: The Vision Can Define Users’ Expectations**

Defining characteristics of a vision shapes the expectations of users of its prototypes to a degree where its users suspend their knowledge and critical thought. Webstrates is a shareable and distributable platform by being accessible in the web browser and by synchronizing the DOM with the Webstrates server. It aims to make the process of creating collaborative applications easier. The seemingly magic mechanisms of synchronizing the DOM led participants to believe that Webstrates is taking care of *everything* related to collaboration—even technically competent users suspended their reflections of what is technically possible. For example, some participants thought that the contents of the `<canvas>` element would be synchronized by Webstrates, even
though they knew that these contents are not reflected in the DOM and, therefore, are not synchronized by Webstrates.

Related to L3, there were participants that were vocal in their criticism of the introduction of new concepts and tried as much as possible to develop software according to their existing knowledge and skills. On the other hand, we saw participants that immersed themselves in the vision of a computational medium that would ease collaboration and programming, and felt frustration when they had to resort to the traditional way of thinking. Where the middleground is — and if it exists — is unclear. The mix between the traditional and the novel of Webstrates has shown itself to be a double-edged sword.

13.10.6 L6: Users Create Conceptual Blends with Familiar User Interfaces

The user interface of novel platforms has a strong influence over how users form mental models of the system and its core principles. Cauldron has the look and feel of a code editor or IDE and, yet, it lacks a lot of the tools that are common in code editors and IDEs such as Visual Studio Code, Atom, or JetBrains WebStorm: For instance, Cauldron supports neither searching for fragments, nor searching text in multiple fragments, nor does it provide its own debugger. However, participants expected some of these features when using Cauldron and were disappointed when their desired features were not available.

Another issue that we saw, was that users experienced an uncanny valley-like effect where some of our concepts were very similar, yet not completely equal to other concepts: For example, code fragments are a central aspect of the way Codestrates v2 works. For participants, the way they are displayed in the tree browser and how they contain just regular code, evoked a connection to files in a file system. However, by making this connection, participants consequently struggled to find a connection for the auto-update functionality of fragments, or that the order of fragments reflects the order of execution. These misalignments, in turn, can cause breakdowns.

13.11 CONCLUSION

Realizing visions of new types of software in a practical implementation that can be used by users can spark tensions between following the vision truthfully and overcoming technical limitations to create a running system. We revisited these tensions for our effort of realizing the vision of computational media in the Webstrates and Codestrates platforms. We used these tensions to investigate how they affect users using the Codestrates v2 and Cauldron platforms. Our results showed us that these tensions also influence how users interact with the system and that divergences from the original vision can cause confusion, breakdowns, or even frustration among users. Synthesizing these re-
sults into six lessons learned, we could uncover themes that creators of new types of software need to take into account when creating practical implementations that diverge from an original vision—both due to technical limitations or pragmatic decisions.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 740548) and from Carlsbergfondet (grant agreement No CF17-0643).
VARV: REPROGRAMMABLE INTERACTIVE SOFTWARE AS A DECLARATIVE DATA STRUCTURE*

—

MARCEL BOROWSKI, Aarhus University, Denmark
LUKE MURRAY, MIT CSAIL, United States
ROLF BAGGE, Aarhus University, Denmark
JANUS BAGER KRISTENSEN, Aarhus University, Denmark
ARVIND SATYANARAYAN, MIT CSAIL, United States
CLEMENS N. KLOKMOSE, Aarhus University, Denmark

ABSTRACT

Most modern applications are immutable and turn-key despite the acknowledged benefits of empowering users to modify their software. Writing extensible software remains challenging, even for expert programmers. Reprogramming or extending existing software is often laborious or wholly blocked, requiring sophisticated knowledge of application architecture or setting up a development environment. We present Varv, a programming model representing reprogrammable interactive software as a declarative data structure. Varv defines interactive applications as a set of concepts that consist of a schema and actions. Applications in Varv support incremental modification, allowing users to reprogram through addition and selectively suppress, modify, or add behavior. Users can define high-level concepts, creating an abstraction layer and effectively a domain-specific language for their application domain, emphasizing reuse and modification. We demonstrate the reprogramming and collaboration capabilities of Varv in two case studies and illustrate how the event engine allows for extensive tooling support.

14.1 INTRODUCTION

It has long been acknowledged that most programs are written not by professional software developers but rather by end-users [167] who,
Figure 14.1: Varv Examples: (a) A todo list web application that is inherently extensible. Here, a basic todo list is extended with the ability to complete and delete todos by adding two new concept definitions and new modified template definitions. (b) A board game toolkit that defines abstractions for board game logic. The games “Checkers” and “Othello” were implemented with the toolkit and then merged into a new “Checkers-O-Thello” game with the addition of a short concept definition. As Varv applications are represented as data structures, higher-level tooling can be developed including a block-based editor (right), an inspector to go from an element in the view to the corresponding template or data (context menu to the left), and a data inspector for live editing application state (middle).

for instance, regularly build small computational tools such as creating their own interfaces in spreadsheets. However, most application software is constructed, packaged, and shared as hermetically-sealed turn-key products [146, 235]. As a result, end-users—including professionally trained programmers—have little power to change the applications they use. In their foundational 1977 piece on Personal Dynamic Media, Kay and Goldberg envisioned software being malleable, so that users could easily redefine and reshape it to suit their idiosyncratic needs [152].

Today, software malleability primarily occurs through scripting (e.g., macOS Automator, iOS Shortcuts, or IFTTT) or add-on extensions (e.g., Firefox, Figma, or Visual Studio Code). While such facilities can yield “customization ecosystems” that increase the value of the application for all users [126], these approaches present a non-trivial burden for both software creators and end-user programmers. Writing extensible software is an explicit choice that software creators must make, and requires careful design and architectural decisions that are often untenable for small-scale software creators to consider as the customization ecosystem is not guaranteed to flourish. Moreover, the range of customizations these facilities support is circumscribed by the design of their APIs, thereby presenting a catch-22: it may not be possible to customize particular aspects of an application if the creator did not foresee the possibility of doing so. Finally, the APIs themselves are idiosyncratic and application-specific. As a result, it can be challenging to engage in customizations in a cross-cutting fashion—for instance, porting an extension from one context to another typically amounts to rewriting it from scratch, and extending or com-
posing add-ons together is inconceivable without direct modification of their source code.

In response, we present Varv, a declarative language for reprogrammable interactive software that decouples specification (the what) from execution (the how). With Varv, users can focus on specifying interactive applications as compositions of concepts, or individual units of dynamic functionality. Concepts comprise a schema, that specifies the shape and type of the concept’s state; actions, that describe valid transformations of the state; and, triggers, or events that cause transitions between states. Inspired by Vega [273] and Vega-Lite [272], concepts are specified as data structures expressed in JSON (JavaScript Object Notation). The Varv runtime is responsible for all execution concerns, including parsing declarative specifications, assembling the corresponding dataflow graph, and handling event creation and propagation. The runtime also handles bookkeeping associated with storing application state and rendering the resultant interface — Varv is designed to be agnostic to the specific ways these processes occur. As a result, Varv can target a variety of data backends or frontend modalities.

Varv’s structured, declarative approach contrasts existing methods for constructing interactive software, which typically involves writing unstructured blobs of imperative code. It yields an accretive development process, with applications that are inherently extensible. In particular, application developers need no longer write explicit extensibility APIs. Instead, to introduce a new piece of functionality, end-user programmers introduce a new JSON object at runtime. These JSON objects can extend or override existing concept definitions in a straightforward fashion or use a series of composition operators to construct new concepts from existing parts. The Varv architecture consolidates new and existing specifications and hot swaps them to produce a live programming experience (i.e., users see changes they make to Varv program specifications reflected immediately). In this way, Varv blurs the boundary between developing the “core” application and extending it, making it possible for users to tinker with interactive functionality incrementally.

We evaluate the feasibility and expressivity of our approach through demonstration [179]. We first instantiate Varv in Webstrates [162], a web-based environment that provides persistence and real-time synchronization of application data (i.e., state). While Webstrates provides our data layer, Codestrates [49] provides a code editing layer on top of web pages that enables instantiating IDE-like tools inside a web app. With this Webstrates-based implementation, we develop two case studies to demonstrate that Varv can be used to author a rich design space of interactive applications. The case studies

---

1 The Swedish word *varv* carries the meanings “revolution” or “in layers.” In geology, a *varv* refers to the annual sedimentary layer in a glacial lake. In the same vein, application code can be layered in our Varv system.
illustrate the experience of using Varv as a live programming tool for user interfaces, akin to real-time manipulation of HTML and CSS using the browser’s built-in developer tools, but now for interactive behavior as well. We show how users can author Varv applications incrementally—one feature at a time, where each feature is implemented as an extension to the application rather than a modification of existing source code—and how concepts can be used to prototype domain-specific languages for developing and composing classes of applications. This implementation required no modifications to Webstrates itself. The case studies also demonstrate the synergies between the two paradigms: using Varv with Webstrates yields a live and collaborative reprogramming experience. However, to illustrate that Varv is agnostic to data storage, we develop two additional prototype implementations, one packaged in Electron [243], an environment that enables local development of Varv applications, and one deployed on Observable [236], a web-based notebook environment for JavaScript.

We, moreover, demonstrate the implications of our approach through a series of prototype sketches of higher-level tooling to support Varv application development. Although Varv applications are specified as JSON-based data structures, we show how this representation facilitates a range of authoring experiences, including visual block-based editing and alternative specification formats such as YAML. Similarly, we show how Varv’s declarative representation enables visual inspectors for debugging. With Varv, we set a foundation for malleable software to enable users—who are, for now, proficient in programming—to modify their software and envision how it is structured. Further, declarative representations can facilitate future research on more usable, higher-level systems for reprogrammable interactive software.

14.2 THE VARV LANGUAGE

14.2.1 Design Goals

Varv’s design is motivated by the following three design goals:

provide a structured declarative representation. Declarative representations have become widely adopted in various domains because they allow users to focus primarily on composing domain-specific primitives at a higher level of abstraction while deferring execution concerns to the underlying architecture or runtime [130]. Varv uses declarative language constructs to define application state and state transitions and provides an execution engine that parses declarative Varv specifications to produce an interactive application. Moreover, inspired by declarative representations of interactive visualizations like Vega [273] and Vega-Lite [272], Varv embeds its
declarative representation of interactive software as a data structure expressed as JSON. In doing so, Varv lowers the threshold for programmatically reasoning about the semantics of interactive software. As a result, it becomes more feasible for the Varv architecture to hot swap declarative specifications to enable live programming and an ecosystem of higher-level development tools to flourish (akin to the one found around Vega and Vega-Lite [301, 313, 323]).

**ACCRETIVE EXTENSIBILITY.** Developers currently rely on extensibility APIs written by software creators to extend interactive software. However, such an approach presents a catch-22: it can be difficult, if not impossible, to customize an application in a particular way if the creator did not design a corresponding API. In contrast, to reduce a creator’s burden of explicitly designing for extensibility, Varv defines interactive applications in terms of individual units called concepts. To extend a Varv application, a developer need only add a new specification to the runtime with entries to augment or override properties of existing concepts or introduce new concepts from a combination of existing parts. Thus, the extensions are themselves units that layer on top of the base definition of an application. This incremental approach facilitates experimentation: a developer can safely try implementing new features, or an end-user can selectively enable or disable extensions without fear of breaking or changing the original program. Moreover, this process of extension-by-addition simplifies resolving conflicting extensions by adding another specification to resolve the conflicting properties. Hence, our aim is the open authorial principle [22] that states that program modification should be possible purely through composition without rewriting existing code.

**DECOUPLE APPLICATION LOGIC FROM INTERACTION MODALITY.** Existing methods for specifying interactive behaviors — namely, event callbacks — tightly couple an input event (e.g., mouse clicks, keypresses, swipe gestures) with the resultant action it triggers (e.g., selecting a piece on a board game, moving it from one square to another). As a result, retargeting an interactive application from one modality to another (e.g., desktop to mobile) or supporting custom interactive triggers (e.g., keyboard shortcuts) requires significant manual development effort. Varv decouples these two pieces: an application can be defined in an abstract, purely self-contained manner with custom, semantically-meaningful event names taking the place of low-level input events (e.g., pieceSelected instead of click). A subsequent specification then makes this abstract definition more concrete by binding semantic events to a specific interaction modality (e.g., pieceSelected is triggered by a tap).
14.2.2  Language Primitives

Concepts (see Figure 14.2) are Varv’s core building block. They define individual named units of interactive behavior — for example, an “item” in a todo list that can be assigned or marked as completed, or a “piece” that can be moved along the squares of a board game or jump over other pieces. Each concept comprises a schema, which determines the concept’s state (i.e., data), and actions, which enumerate the ways this state can change through interaction. Concepts can be augmented or extended in two ways: additional specifications can be introduced (e.g., Figure 14.2b) which reference an existing concept by name, and extend or override its properties; or, a variety of extension operators can be used to define new concepts from existing parts.

Varv’s concepts combine ideas from several different programming paradigms. At first glance, concepts seemingly map to classes in object-oriented programming (OOP), offering a mechanism for modularity, reuse, mixins, and traits. However, Varv’s concepts make a fundamental departure: concepts are not encapsulated units (i.e., a concept’s state and actions can be referenced from another). This design choice emulates the Store design pattern adopted by many popular JavaScript frontend libraries (e.g., Redux, Vue, and Svelte). Stores centralize application states, decoupling them from state transitions to aid rapid prototyping and developing cross-cutting components. Varv’s unencapsulated concepts retain this affordance without sacrificing the modularity of OOP classes. We elaborate on these and other differences between Varv and existing programming paradigms in Section 14.7.3.

14.2.2.1  Schema

The schema defines the shape and type of data associated with a concept. The syntax for the schema definition uses a modified version of JSON Schema [251] and supports a subset of the JSON Schema functionality. Varv extends JSON Schema with shorthands, making the language more concise and easier to read and write. For example, `{"label":{"type":"string"}}` can be written as the shorthand `{"label":"string"}`. Varv concepts can be referenced directly by name within the schema to specify nested state. For example, Figure 14.2 defines the schema of a "todoList" concept as an array of "todo" concept instances. Varv also supports deriving properties from existing state by specifying a "derive" object which expects an array of "properties" that are processed through an array of "transform" actions. Varv merges the properties and actions to generate a function that produces the derived value. For instance, in Figure 14.2, the "totalCount" property of the "todoList" concept is calculated as the "length" (a built-in action) of the "todos" property.

Early prototypes of Varv did not provide an explicit definition of concept state. Instead, the state was implicitly created and manipu-
14.2 The Varv Language

(a) A concept definition that is abstract as it does not reference specific interaction modalities.

```json
C concepts: {
  todoList: {
    schema: {
      todos: { array: "todo" },
      completedCount: "number",
      totalCount: { "number": {
        derive: {
          properties: [ "todos" ],
          transform: [ { length: "todos" } ]
        }
      }
    },
    actions: {
      updateCompletedCount: {
        when: [ { action: "toggleCompleted" } ],
        then: [...]
      }
    }
  },
  todo: {
    schema: { text: "string", completed: "boolean" },
    actions: {
      toggleCompleted: {
        when: [ {{ click: { view: todoCheckbox } } }]
      }
    }
  },
  assignable: {
    schema: { assignedTo: "string" }
  }
}
E extensions: {
  join: [ "todo", "assignable" ],
  as: "assignableTodo"
}
```

(b) Extending the abstract specification with concrete references to modality-specific input events (the toggleCompleted semantic event, defined in the abstract concept, is triggered when the todoCheckbox widget is clicked).

```json
C concepts: {
  todo: {
    actions: {
      toggleCompleted: {
        when: [ {{ click: { view: todoCheckbox } } }]
      }
    }
  }
}
```

Figure 14.2: The components of a Varv concept definition for a simple todo list. As a convention, and to demonstrate the merging of concept definitions, we split the definition into an abstract and a concrete part. The abstract part provides definitions for a todoList, a todo, and an assignable concept. Each concept has a schema and the todo concept has an action which encodes a state transition (omitted) in a then-block. An extension is used to create an assignable todo by joining the todo and assignable concepts. The concrete part binds the toggleCompleted action to an interaction specific to a DOM view using a when-block. (Quotation marks from JSON keys removed for readability.)
lated through sequences of actions. However, as we built increasingly complex applications, we discovered that this implicit treatment reduced visibility [37] into concept state (i.e., it was not clear what properties were available for access on a given concept). In contrast, by explicitly enumerating a concept’s properties and their types, Varv schemas help formalize concept state. They serve as a baseline level of documentation for the structure of concepts within the program, and types are validated at runtime to reduce error-proneness [37]. Schemas, moreover, aid concept reusability. For instance, in early prototypes, Varv stored concept state directly on DOM nodes. This approach introduced hidden dependencies [37], making it challenging to adapt concepts to new contexts without introducing knock-on effects to the output interface. It, similarly, introduced a premature commitment [37] by requiring every concept to be reified as an interface element. In contrast, with schemas, concepts can be reasoned about in purely abstract ways and referenced throughout a declarative specification without being mapped to a concrete user interface component.

14.2.2.2 Actions and Triggers

Actions provide a common abstraction for specifying state transformations, and consist of two parts: an optional when-block and, a required then-block.

The when-block defines an array of triggers or events that cause the action to be executed. Varv provides two types of triggers (see Appendix C.3). Reactive triggers govern concept space: they fire when a concept’s state changes, or when a concept’s action finishes executing, or at a given interval. For instance, in Figure 14.2a, the updateCompleteCount action makes use of a reactive trigger — this action executes once the toggleCompleted action of the todo concept has run to completion. View triggers, on the other hand, fire when input events (e.g., mouse clicks or key presses) occur. For example, Figure 14.2b demonstrates how an additional specification can bind purely abstract concrete definitions to concrete interface elements using view triggers — the toggleCompleted action of the todo concept fires when the todoCheckbox element is clicked.

The then-block specifies an array of actions that should be executed. Nested actions can include either other concept actions or Varv’s primitive low-level actions (see Appendix C.2). These built-in actions include operations for manipulating a concept’s state (e.g., arithmetic calculations, string and array manipulations, etc.) as well as determining control flow (e.g., early exiting a chain of actions, or forking the chain to execute an independent action). This design allows for recursion (i.e., an action can call itself within the then-block), with a “where” control flow action used to indicate the terminating condition. The output of an action can be referenced using the dollar sign — by default, the output is named for the action (e.g., $length
the output of an upstream "length" action) but these variables can be renamed using the "as" property offered on many actions. Finally, actions can be parameterized using the using the @-symbol in front of parameter names, e.g., "@newTodoLabel". These parameters can subsequently be provided as properties when referencing the action downstream. The addNewTodo action shown in Appendix C.1.1 provides a complete example of these ideas. When it is executed, it creates a "new" instance of the todo concept using the value provided by the newTodoLabel parameter (populated on line 45). The output of this action is stored in the $newTodo variable (due to the "as" property specified on line 25), and is used to append to the list of todos.

Concept actions do not need to define both blocks. Rather, concept actions can be directly defined as a then-block (bypassing the nested format) and additional, separate specifications can later bind actions to specific interface elements. For example, Figure 14.2 uses this convention to first define the abstract idea of a todoList comprised of todos which can be completed (Figure 14.2a). Note, the toggleCompleted action does not define a when-block. In a subsequent specification (Figure 14.2b), this action is bound to click events that occur on the todoCheckbox element. By following this convention, a concept action can serve as an abstraction for a sequence of nested actions, and helps decouple the application logic of an interactive component from a specific reification or modality.

Varv also allows users to register custom actions written in Java-Script (see Appendix C.1.4). Custom actions can extend the Varv standard library with additional functionality, integrate Varv with existing JavaScript code, or let users write complex business logic using imperative code.

14.2.2.3 Extensions

Extensions are mechanisms that enable the reuse of concepts. Out of the box, Varv supports merging and overwriting properties using naive declaration merging based on JSON keys. Figure 14.2 uses declaration merging to extend the toggleCompleted action on the todo concept with a when-block. However, during our prototyping process, we quickly realized that naive declaration merging is limited to only extending or overwriting existing concepts. In particular, there is no way to use naive declaration merging to build higher-level concepts that are ad hoc compositions of existing concepts.

To support more nuanced mechanisms for concept reuse, Varv offers four extension operators: "inject", "join", "omit", and "pick". "inject" merges the definition of one or more source concepts into another target concept. The source concepts are left unaltered while the target concept gains new functionality. The "join" operator is similar to "inject" but merges one or more source concepts to create a new concept, leaving source concepts unaltered. The "omit" operator
takes a source concept and can remove actions and schema from
the concept, altering the source concept, providing a mechanism to
remove functionality via addition. The "pick" operator takes a source
concept and selects a subset of the schema and actions to create a new
target concept, leaving the source concept unaltered. Using these four
operators, users can define a library of concepts as mixins and inject
them into other concepts to prototype applications rapidly.

14.2.3 Event Flow

Varv is an reactive and event-based system. Events in Varv are data
objects that are used to transfer information. Events are emitted
from triggers, passed on to actions, and then terminate once an
action is performed.

![Event structure in Varv](image)

**Figure 14.3:** The structure of an event in Varv.

14.2.3.1 Event Contexts

Events contain contexts and shared variables (see Figure 14.3). A context
is also a data object that consists of a concept instance, the target, and
variables in the context. The target is required by many actions to
define on which concept instance an action should work. For example,
consider a todo concept that contains the string property text. The
action {"length": "text"} computes the length of the text property.
In order to know from which instance the action should take the text
property from, the target is used. Once the action is performed, the
"length" action adds the variable length with the result to the context
variables of the respective context.

An event can contain multiple contexts, because actions might
need to work on multiple instances at once and do something for
each of them. This is inspired by JavaScript array methods such as
map [195]. The contexts of an event can be modified by actions, e.g.,
the "select" action replaces the current contexts in an event with one
context for each concept instance the selection defines. Other actions
also enable to remove contexts from an event, e.g., the "where" action
filters contexts based on the properties of targets or variables in the
context. Figure 14.4 shows an example where first the "select" action
is used to select all todo concept instances, then the "length" action
is used to retrieve the length of the text property of a todo, and lastly the "where" action is used to filter the one with a length of less than four characters.

14.2.3.2 Event Creation and Passing

Events are created by triggers. When creating an event, a trigger can add contexts to a new event, for example, the "click" trigger adds the concept instance of the element the user clicked on—if it is a concept instance—as a target and the coordinates of the mouse click as variables.

Events are by default passed from one action to the next, each working on the same event. This, however, can lead to changes to the variables or contexts of an event. If an action should be performed without affecting the event, the "run" action can be used. This effectively makes it possible to split the event up. If an action removes all context from an event, by default, an empty event without contexts is passed on to ensure the execution of consecutive actions. In this case, however, the event would lose all its context variables. To prevent this, Varv stores variables that are the same across all contexts in the shared variables. These are persisted even if no contexts are in the event anymore and added back to context variables once new contexts appear. If an event should not be passed on if there are no contexts
left, actions like "select" or "where" have an option to stop the event, allowing them to act as a gate.

14.3 THE VARV ARCHITECTURE

The overall architecture of Varv consists of six main components: the event engine that reads in concept definitions, templates that define how these concepts are rendered in the view layer, and mappings that define where data from concept instances should be stored in the data layer (see Figure 14.5). This section summarizes the purpose of each of these components; their implementation in our Varv prototype is described in Section 14.4.

14.3.1 Concept Definitions and the Event Engine

Concept definitions are files that use the concept language which was introduced in Section 14.2.2 to define the interactive behavior of an application. There can be any number of concept definition files in a Varv application. All concept definitions are merged by the event engine at runtime. When being merged, concept definitions later in the document overwrite earlier ones — i.e., existing concept, actions, and properties can be added, suppressed, or overwritten by adding new concept definitions at the end of a document.

To illustrate this merging process, we used the convention of splitting concepts into two parts in our examples: an abstract part, that contains actions that are view-agnostic, and a concrete part that contains actions that are view-dependent. By separating these parts, it is possible to reuse the core logic of a concept if another view is targeted.

![Figure 14.5: Architecture overview of Varv.](image-url)
14.3.2 Templates and the View Layer

The view layer contains views and templates. A view is a component that renders a user interface with which users can interact, for example the DOM. Making the view independent from the event engine, allows it to connect different types of views to the same underlying interactive behavior and state of an application.

Templates are used to specify how state should be represented in the view by referring to concepts and properties in them (see Appendix C.1.3 for an example). A template is view-dependent, thus, different views require different templates. In the DOM, for example, a template could be written in HTML while in other views they might be required to provide a scene graph or other structures. The view then combines these templates with the state it retrieves from the event engine to generate a user interface. By generating the user interface in this way, elements in the view can be connected to their underlying concepts and state, allowing for higher-level tooling such as a view inspector (see Section 14.6.2). Lastly, views can also add view-dependent view triggers, which can be used in actions in the concept definitions to react to user input in the view.

14.3.3 Mappings and the Data Layer

The data layer contains data stores. Data stores allow Varv to store state of concept instances and their properties in them. A data store can be anything that can store data in a key-value format. One purpose of using a separate data layer in Varv is to be able to dynamically store data in heterogeneous ways, which allows users to define properties in the schema of concept definitions without having to take care of how and where it is stored. Another reason for using a data layer is to decouple the state of an application from the interactive behavior. This, for instance, enables to hot swap concept definitions in the event engine or to connect different application to the same data store. The latter allows users to create their own personalized applications, but still being able to collaborate on shared data (Section 14.5.1 demonstrates this).

Mappings are pointers that define in which data store state is stored. Mappings can be defined for each property of a concept. This allows, for example, to store ephemeral state like the content of an input field in a data store that is not shared with other users. If properties are mapped to multiple data stores, Varv synchronizes state between all selected data stores. Data stores can, further, notify the event engine about updates to the data, for example, if a remote user changes data in a shared data store. The event engine then synchronizes the data with other data stores and notifies actions and the view about the change.
14.4 IMPLEMENTATION

Our main implementation of Varv\(^2\) is written in JavaScript, builds on top of the Webstrates [162] platform and the Codestrates v2 [49] framework, and runs purely client-side in a web browser and uses Codestrates v2’s extensible in-app IDE Cauldron for development (see Figure 14.6a). This section will first describe the Webstrates platform and Codestrates framework and what parts are used for Varv. Then we explain how the control flow of Varv works and how we achieve live extensibility.

We have also implemented a proof-of-concept version of Varv that is independent of Webstrates. We use this version of Varv to package Varv applications as Electron [243] apps using regular JSON and HTML files stored on the disk for concept definitions and templates (see Figure 14.6b).

We, additionally ported this version of Varv to Observable [236] using tagged templates [196] for concept definitions and templates. This makes it possible to use the computational notebook view of Observable to create, share, and incrementally develop Varv applications (see Figure 14.6c). Further, this demonstrates the portability of the Varv runtime to contexts outside of Webstrates.

14.4.1 Building on Webstrates, Codestrates v2, and Cauldron

14.4.1.1 Webstrates

Webstrates [162] is a software platform for building reprogrammable, collaborative software on the web purely from the client side. The simple yet powerful mechanism behind Webstrates is to synchronize and persist changes to the DOM of a web page served from the Webstrates server. This includes changes to embedded code (JavaScript, CSS, and more), effectively making it possible to both collaborate on using and programming software. As default the whole DOM is synchronized, but to support a relaxed WYSIWIS (What You See Is What I See), a custom <transient> element can be used to create subtrees that are not synchronized — e.g., for UI elements.

14.4.1.2 Codestrates v2 and Cauldron

Codestrates v2 [49] provides a model for controlling the execution and interdependence of scripts of various types.\(^3\) Furthermore, it provides

\(^{2}\) Varv on GitHub: https://github.com/Webstrates/Varv
(Retrieved February 22, 2022)

\(^{3}\) While this model enables executing JavaScript code at runtime, Codestrates v2 does neither handle duplicate event listeners, other issues that come up when re-executing imperative code at runtime, nor synchronize runtimes across clients. Hence, limiting its use for live and collaborative programming.
(a) The main implementation of Varv. It builds on top of Webstrates. The Cauldron editor can be opened in the web browser.

(b) A proof-of-concept implementation of Varv in Electron.

(c) A proof-of-concept implementation of Varv in Observable.

Figure 14.6: Screenshots of our implementations of Varv.
an API for instantiating code editors for specific scripts (stored in so-called code fragments) in the user interface. Codestrates v2 is bundled with its own extensible development environment Cauldron, which allows within-application modification: users are able to create, edit, and run code fragments directly inside the web browser without additional software (see Figure 14.6a). Codestrates v2’s execution engine can be used independently of Webstrates (e.g., as in our Electron prototype).

14.4.1.3 Varv

Varv adds a new Codestrates v2 fragment type for concept definitions. Templates are stored in HTML fragments and styling in CSS fragments. Varv leverages the synchronization with the Webstrates server to synchronize state that is stored in the “dom” data storage — enabling collaboration. Varv inherits the ability to edit code directly in the interface, collaborate in real-time, and version both data and code from Webstrates. Concept definitions, templates, and the concept data store are all persisted in the DOM in custom tags hidden from the browser view using CSS. The user interface generated from Varv is wrapped in a transient element, hence synchronization of application state only happens through the data storage.

14.4.2 Event Engine

14.4.2.1 Building and Rebuilding the Model

The event engine queries all concept definition fragments and parses their JSON code. Concept definitions are merged sequentially into a single definition. Extensions to the concepts such as injections are

![Figure 14.7: The software stack of two of our Varv prototypes. Our main Webstrates-based implementation uses Cauldron as its editing environment with Varv-specific tooling built on top. The Electron-based prototype uses Codestrates for code execution but is independent from Webstrates. Electron is used to store and load code from the file system. Code can, e.g., be edited using Visual Studio Code, which could be extended (not implemented in our prototype) with Varv support by using JSON Schema or by porting our block-based editor.](image-url)
performed after the merge in the order they appear in the concept definitions. When a new concept definition is added or any of the existing ones are changed or deleted, the running model is destroyed and a new model is built. Application state is not lost as it is stored separately in data stores.

The merged model contains all concepts, schema, actions, mappings of properties, and data stores defined in the concept files. Once merged, the engine uses the mapping and data store information to connect properties of concepts to their mapped data stores and notifies the data stores of their connection. Afterwards, the view is notified of the updated model. Lastly, the engine subscribes each action that has a when-block to their respective triggers.

Primitive triggers register themselves in the event engine once instantiated. Once the trigger fires an event, which consists of a string containing the trigger name and a JavaScript object containing the context, the event is passed to the event engine that distributes the event to the actions subscribing to that trigger. Like triggers, primitive actions register themselves in the event engine once instantiated. Once an action is triggered, its actions are executed: Each action receives the list of contexts in the event and the action options defined in the concept definition (see Section 14.4.3 for more detail on the event flow).

Actions with the same name can be defined in multiple concepts, thus, we provide a look-up function to find the correct action. To target a specific action implemented in a concept, a dot-notation can be used, for example, "checkers.markValidSquares" or "othello.markValidSquares". There is a lookup order starting first with primitive actions to searching for actions with a given name in any concepts in the model.

14.4.3 Data Stores

Types of data stores are registered in the event engine like triggers or actions. They can be used to create custom named data stores in concept files. Our implementation of Varv defines three types of data stores: "dom", "localStorage", and "memory". By default, properties are mapped to the "dom" data store, where they are persisted and synchronized with other clients through Webstrates. An option for the "dom" data store can change the location for storing the state in the DOM to another webstrate, allowing multiple applications to work with the same data. Properties can also be stored in "localStorage" or "memory" data stores, if they are ephemeral or should not be shared with other clients. Our Electron-based prototype uses the "localStorage" data store for persistence.

Once the event engine has loaded the model, it connects itself to the data stores defined in the model. Next, it maps each property to the data stores that it is mapped to and registers "getter" and
"setter" callbacks of the data stores in the property. After registering the callbacks for each data store, the event engine attempts to load already existing data of a property from each data store and publishes it to all other data stores of that property and, hence, synchronizes them. If a property is mapped to multiple data stores that contain conflicting data, the ones first in the list of mappings overwrite the data of later ones.

If data changes outside the Varv system, e.g., remote changes to the "dom" data store, a data store can notify the event engine about changes to properties, which will then synchronize it with other data stores and notify views and the "stateChanged" trigger. Changes to properties from actions or views are sent to the event engine and forwarded to the registered data stores.

14.4.4 Views

Views exist mostly independent from the rest of the Varv system. They can connect to the event engine and register to concepts and properties to get and set properties, as well as register their own triggers. Our implementation includes the "dom" view that renders data in the DOM of a website.

It parses all <dom-view-template> nodes and collects what concept instances are required by the templates. Next, it subscribes to these concept instances in the event engine and retrieves a list of concept instance objects with references to their properties. The view is notified on updated properties, created or removed concept instances, and is able to set changes of properties — again, passing new values to the event engine, which forwards it to the data stores. If any template changes, the view unsubscribes all properties and repeats the process. In the templates, the "dom" view looks for special attributes (concept, property, and value) and replaces curly braces of properties in other attributes or text nodes with the values of properties (see Appendix C.1.3). Additional style can be added using CSS and assets like images or icons can be uploaded to a webstrate using Cauldron. The "dom" view supports mouse and keyboard events as view triggers.

14.5 Case Studies

To illustrate how Varv applications enable new ways of extending and modifying software, we present two case studies of how Varv can be used: The first case study illustrates how an existing todo list application can be collaboratively modified through addition of code. The second case study demonstrates how Varv can be used to create a declarative abstraction layer for board games and how different applications can be built using these abstractions. The case studies are also presented in the paper’s accompanying video.
addition to the two case studies, we briefly describe other application examples we explored.

14.5.1 Case Study 1: Todo List

Imagine two computer science students, Melissa and Daniel, who work together on a course project. To manage their tasks, they use a simple collaborative todo list web-app created in Varv. During the first half of their project they work tightly together. However, they increasingly need to split up tasks and work on them in parallel. They now want a feature in the todo list that lets them assign todo items between them.

**Adding the “assignee” field.** To modify the todo list they click an “Edit” button in the top right corner of the interface to open Cauldron. There is a list of files that includes the concept definitions for the app: "todo", "todoList", and "todoInput". They need to modify the "todo" concept. Daniel creates a new folder with the name “assignee” with a new concept file. He adds the new property \{"assignee":"string"\}, which stores the name of the person responsible for the todo item (see Figure 14.8a). Next, he needs to show that information in the view, so he also creates a new template file. He copies over the template from the original todo item and adds a line with an input field for the property. While doing so, he can immediately see the input field appear in his view and test if it works by writing his name into the input field.

**Adding filtering.** Melissa’s friend Samantha has written a filtering mechanism for the todo list. Melissa can add the filtering to their app by dragging the folder containing a concept, template and style file, from Samantha’s in-app IDE Cauldron to hers. Daniel wants filtering on his own "assignee" field as well. However, he does not really understand the filtering code so he asks Melissa for help. Together, they try to understand the code and Melissa adds some code to the filtering to also support the assignee filtering. While Melissa is coding, Daniel has the app open, and he can immediately try out the effects of code on the app.

**Using separate views.** During the second half of their course, Daniel adds more and more features to the todo list. Melissa finds the interface cluttered and wants a simpler app. So, she creates a copy of their todo list. In the copy, she deactivates all the features she does not want in her version of the todo list and — because she is making changes anyway — also adds a dark theme for the web-app. To still be able to work on the same data as Daniel, she remaps the data store of the new app to Daniel’s (see Figure 14.8b).
Daniel adds a new concept definition to the todo list app and adds the "assignee" property. Once the concept definition is activated, the property is immediately added to todos, seen in the inspector in the center bottom of the screen.

(b) Daniel and Melissa can both use their preferred view and functionality in their app. While Daniel (left) uses more features and a light theme, Melissa (right) uses a more simple layout with a dark theme. The underlying data is shared.

Figure 14.8: Screenshots of the first case study.

**How it Works.** Varv supports incremental application development, thus, Daniel and Melissa can add functionality step-by-step. Adding new concepts or templates allows them to overwrite the parts of an app that they want, without having to change the original implementation. This is enabled by the event engine merging all concept definitions and rebuilding the model after every change. This, further, enables them to add new functionality from their peer Samantha without having to touch the code of the original todo list or their assignee feature. As they are adding new functionality accretively in new concept definitions, it is also possible to go back to prior versions by disabling these definitions in Cauldron.

With Varv running on Webstrates, they can collaborate on the code of the app in Cauldron and test new functionality together. Varv
makes the collaborative testing possible by automatically reloading concept definitions whenever changes are done locally or remotely. As the todo app is stored in a webstrate where the app is self-contained, i.e. both the data and the application code are stored together, they can generate copies to create personalized applications. Decoupling the interactive behavior (concept definitions) and the view from the data, further, makes it possible to remap the “dom” data store to another webstrate, providing means to create customized views while using the same data.

14.5.2 Case Study 2: Board Game Toolkit

Sean is a fan of board games like Checkers⁴ or Othello.⁵ He has ideas for modifying existing games to make them more attractive and wants to realize some of them as web apps to play with friends.

BUILDING A TOOLKIT. Sean wants to make a toolkit for games in Varv so he does not have to build new games from scratch. He starts by creating a "game" concept for more general game mechanics like taking turns and who the winner of a game is. Next, he creates the basic concepts of board games: the "piece" and the "square". Both share common traits: they have one of two colors and a location defined by a row and a column. Sean creates a shared mixin with helper actions for each of those and calls them "colorable" and "locatable". He injects both mixins into both concepts. In order to let players select and move pieces, he adds another mixin that he calls "markable", which enables him to mark pieces or squares. The mixin contains actions to, for instance, "mark" a piece or check whether a piece "isMarked".

CREATING CHECKERS AND OTHELLO WITH THE TOOLKIT. Next, Sean adds a new concept for the checkers game, where he adds actions that are specific to Checkers, for example, to handle when a piece jumps over another piece. In doing so, he uses the actions from the concepts he created in the toolkit. Sean shows the game to his friend Amy. She wants to implement a game by her own. Amy makes a copy of Sean’s game and disables the Checkers concept file. She then creates a new concept file for her favorite board game Othello. After implementing the Othello game, she immediately tries out the game in an online multiplayer match against Sean.

MODIFYING THE GAMES. Sean plays around with variants of his Checkers game. He makes concept definitions that he can toggle on

---

⁴ Checkers or Draughts: https://en.wikipedia.org/wiki/Draughts (Retrieved February 22, 2022)
and off with small adjustments, which, e.g., enables pieces to move both forward and backwards all the time—making the game more complex to play. In yet another variant, he lets players have two consecutive turns after each other. For a final variant, Sean wants to combine the game rules of Amy’s Othello game with Checkers. He asks Amy to join him remotely in creating their own “Frankenstein-game” Checkers-O-Thello. They add a new concept definition, where they resolve issues between the game rules of both games. After some fixes, they activate the game rules of both Othello and Checkers and can now use Othello’s game rules for placing pieces and Checkers’ game rules for moving pieces.

**How it works.** Varv lets users create their own abstractions over complex state transformations in the form of custom concepts and actions. Sean leverages this by creating concepts for pieces and squares and by adding meaningful actions to them. By doing this, he effectively writes his own domain-specific language for creating board games. Using this language in the Checkers game, he can think about high-level rules of the game, such as “Which are the valid squares a piece can move to if it was selected?” rather than low-level problems like “How do I detect if the when the user picks a piece?” Once created, these abstractions can be reused, so that, for example, Amy can also create her Othello game without having to solve low-level problems first.

This process of modifying games and creating variants of them is supported by Varv’s support for incremental application development. It enables Sean to modify only some actions of the Checkers game in his variants, without having to recreate the whole game several times. When implementing the Checkers-O-Thello game, Varv’s real-time collaboration makes it possible for Sean and Amy to work together on the code, and state synchronization through the Webstrates-based data storage to play the game as a multiplayer game. As both of their games were created using the same abstractions, merging them is a straightforward task. They need to add a few actions to their game implementations to get the game rules of both games work together in a single game. By accretively adding these actions, they do not even have to touch the concept definitions of the two already existing games—something that would be difficult to do in conventional imperative programming languages.

### 14.5.3 Other Examples

Besides the two case studies, we also explored creating other types of applications with Varv:

**UI Designer.** The UI Designer can be used to create mock-ups of user interfaces and the navigation of apps—similar to Figma [99]. It
lets users create multiple screens and add elements such as labels, boxes, or buttons within those screens (see Figure 14.9a). Elements can be moved and resized with the mouse cursor and can link to other screens. In the preview mode, interactions can be tested and used to navigate to other screens by clicking on them.

**Computational Notebooks.** The Computational Notebook is written in Varv and lets users write their own Varv applications using a computational notebook interface. Each cell in the notebook can be a concept definition or a template making it possible to quickly sketch Varv applications (see Figure 14.9b). New cells can be added using buttons in the toolbar. The Computational Notebook also adds a custom action `AddFragment` (similar to Appendix C.1.4), written in JavaScript, which can add new fragments to the DOM.

### 14.6 Tooling

To demonstrate how the architecture of Varv and its declarative language design lends itself to create tooling on top of it, we implemented multiple authoring and debugging tools for Varv. The JSON-based data structures, in which Varv applications are defined, are simple and structured, so other authoring environments can be used to author Varv applications. The decoupled architecture of view, data, and event engine, in addition, facilitate to create inspectors for data and the elements in the view — enabling not only to inspect the view of applications but also their interactive behavior.
By specifying applications in Varv in JSON-based data structures, a common file format in the modern Web, Varv provides a common interface for other authoring tools to connect with. We created three examples of authoring experiences that allow to define interactive behavior in Varv.

YAML-BASED EDITOR. Readability and ease of writing actions could be improved by using YAML instead of JSON as the language for concept definitions. As a superset of JSON, it is possible to add support for specifying concept definitions in YAML instead of JSON. Using indented delimiting, YAML potentially makes writing code easier as less special characters are used (see Figure 14.10a).

JSON SCHEMA AUTO-COMPLETION. We created a JSON Schema [251] specification for the concept language and most of its primitive actions and triggers, and data stores. JSON Schema is widely used and supported by many code editors and IDEs as well as Cauldron. Registering the JSON Schema in these editors enables autocompletion, type checking, and validation (see Figure 14.10b). Autocompletion supports users in exploration, while type checking and validation can help to resolve wrong specifications/parameters while writing the code. A current limitation of the JSON Schema is that it is limited to the primitive actions and triggers, actions that are defined in concepts are currently not added.

STRUCTURED BLOCK-BASED EDITOR. To create a more tangible and explorable authoring experience, we implemented a structured and block-based editor (see Figure 14.10c). The editor is implemented using the Blockly [115] library and provides blocks for most primitive actions and triggers. The sidebar of the editor makes it easy to explore available actions and triggers. Editing concept definitions in
14.6 Tooling

The declarative structure of applications and the decoupling of the engine from the data and the view layer means that the view is generated from the data and the model in the event engine. In doing so, the view can be connected to both the interactive behavior and the underlying data. We show in two inspection tools how this connection can be leveraged to support debugging and testing. By bringing applications and the development environment with their underlying code closer together, we aim to make it easier for users to find the relevant code for their planned modifications, lowering the threshold to modify their applications.

Data Inspector. The data inspector lives inside the Cauldron editor (see Figure 14.11a). In its tree browser, concepts types and their instances can be modified, created, or deleted. Selecting a concept shows its schema and actions, and selecting a concept instance shows its properties and their values in the inspector tab underneath the tree browser. Values can be edited and modifications are directly applied in the view. Creating the data inspector was possible as the information about schema and actions of concepts is available in the model of the event engine in a structured format.

Figure 14.11: Two debugging tools for inspection we implemented for Varv.

the editor automatically updates the JSON, creating a live programming experience. By applying changes immediately to the JSON and hence the event engine, the editor allows for quicker ways to enable and disable actions and experimenting with different interactive behavior. The block-based editor, however, has the same limitation as the JSON Schema, as it currently not dynamically adds actions from concepts as blocks.

14.6.2 Debugging Tools

The declarative structure of applications and the decoupling of the engine from the data and the view layer means that the view is generated from the data and the model in the event engine. In doing so, the view can be connected to both the interactive behavior and the underlying data. We show in two inspection tools how this connection can be leveraged to support debugging and testing. By bringing applications and the development environment with their underlying code closer together, we aim to make it easier for users to find the relevant code for their planned modifications, lowering the threshold to modify their applications.
VIEW INSPECTOR. The view inspector can be used in the "dom" view. By holding the control key and right-clicking on any element in the view, the view inspector shows a menu with information about the clicked element (see Figure 14.11b). The view inspector checks if the selected element or any of its parent elements is an instance of a concept and which template files were used to generate the view. Using the information about the concept instance, the view inspector creates a link to the instance in the data inspector that users can follow to inspect the properties of the selected element.

The view inspector is enabled by the decoupling of concept definitions, data, and the view. As the view is generated at runtime and updated whenever a concept definition or template is modified, it always retains a connection to the concepts and data that were used to generate it. We created the view inspector as a step into breaking up the strict border between the application and the development environment, supporting users in finding relevant code for their modifications.

14.7 RELATED WORK

14.7.1 Declarative Programming

Declarative languages separate the how from the what and allow users to focus on the specifics of their domains [130]. Some would argue that Varv is not declarative because actions written in Varv consist of series of steps and can assign variables. However, computer science literature does not provide a concrete notion of what declarative programming is. Lampson describes a declarative program as a program which has few steps, is a good match for the users view of the problem domain, provides mechanisms for composition, gives big primitives so that users can get a lot done without having to write code, and allows for clean escape hatches so that imperative programming is allowed when needed [174, 253]. Varv meets all of these conditions. Varv provides high level primitives for binding data and updating state, enables rich mechanisms for composition, and allows users to specify custom actions using JavaScript. Additionally Varv provides capabilities for users to develop their own domain specific primitives, enabling greater expressivity in the large domain of interactive web applications.

Declarative languages have become widely adopted in many domains because they make it easier to accomplish complex tasks. Database query languages such as SQL have allowed database developers to focus on describing what data they want while the query optimizer determine how best to get the data using available indexes and joins [159]. HTML and CSS let web developers describe what markup and styling to use while the browser optimizes the page rendering [130, 159]. Vega-Lite lets users describe high level
incomplete visualization specifications and uses heuristics and rules to resolve ambiguities and generate a visual representation which follows visualization best practices [313]. Beyond performance improvements, declarative languages are highly suitable for integration with higher level tools [273]. Within the Vega and Vega-Lite ecosystem Voyager [313], Lyra [271], and Altair [301] have been developed to let users generate visualizations through exploration, direct manipulation, and Python bindings respectively.

Because of the benefits of declarative languages there have been many attempts to write declarative languages for the web. Many of these attempts such as Araneus [218], AutoWeb [109], STRUDEL [97], and WebML [61] provide declarative languages, both graphical and textual, which can be used to derive multi-page websites from various data sources. These projects use multiple approaches for specifying the structure, navigation, and presentation of websites, but are focused on sites where each page is a statically generated view of data rather than an interactive application. SOBL [75] is closer to Varv and provides a declarative specification for user interactions which is automatically parsed into static HTML web pages and state transition diagrams, but does not close the loop and generate interactive applications, or provide mechanisms for composition of existing programs.

Vega and Vega-Lite [272–274] provide mechanisms to allow users to convert a definition of a data visualization, written in JSON, into an interactive chart. Varv is an extension of the same idea to applications. Because of the similarity, Varv uses many similar mechanisms to Vega and Vega-Lite. For example both Vega and Varv allow users to define reusable pieces of functionality by associating the functionality with a name and both Vega and Varv allow users to extend the runtime with user defined functions, while still allowing users to invoke the functions declaratively.

KScript and KSWorld [237] are respectively a scripting language and an editor for end-user authoring of software, that emphasize reduction of accidental complexity and live programming to support exploratory application building. Hence, the project shares similar goals with Varv. Also, similarly to Varv, KScript provides declarative language constructs for event-flow based programming. However, they are embedded in an object-oriented imperative programming language inspired by JavaScript, whereas Varv is a fully declarative programming model.

14.7.2 Alternative Representations of Web Applications

There is a long line of research which attempts to make writing simple web applications easier. Object Spreadsheets [211] identifies that many end-user programmers are familiar with the spreadsheet model, and uses a new computational model for spreadsheets to enable the devel-
development of web applications. However, Object Spreadsheets is focused on providing powerful spreadsheet based mechanisms for data modeling, and provides few abstractions for enabling interactivity, and falls back on imperative scripting for mutating state. Quilt [33] provides a similar spreadsheet backed metaphor for web applications, but provides almost no data abstractions, and acts essentially as an HTML attribute based template language for binding elements on a web page to rows in a spreadsheets. Gneiss [63] provides a live programming environment for developing websites from web data using spreadsheets but does not provide mechanisms for code reuse or composition. Wildcard [188, 189] also uses a spreadsheet metaphor, but enables the augmentation of existing websites with additional data rather than the construction of independent applications. Varv provides a declarative specification for application logic and user interactions, as well as data bindings to a data store. Because Varv represents a declarative target, we believe high-level tooling such as live editing environments or integration with external data sources could be built on top of Varv.

Mavo [302] allows users to develop CRUD applications with a template language built directly into HTML. The user defines a data schema implicitly by adding attributes and expressions to HTML elements, and Mavo provides out of the box support for editing data directly in the interface. The primary goal in Mavo is to allow users to directly manipulate and define the shape of a data schema in a UI layout. Varv also supports a template language but separates the definition of the data model from the template. Varv has less emphasis on direct manipulation of data and instead focuses on composition and malleability of concepts. In addition, by separating data from the view, Varv allows users to write application logic once while targeting multiple view layers. Additionally, because the data logic in Mavo is encoded directly in the layout, creating new layouts while retaining existing data logic can be non trivial.

14.7.3 Software Development Paradigms

14.7.3.1 Object Oriented Programming (OOP)

Varv’s notion of concepts has direct parallels to classes in OOP. Concepts consist of two parts, a schema, and actions. The schema is similar to class properties, and the actions are similar to class methods. Varv’s extension methods — “inject” and “join” — are synonymous with mixins and traits. Because of these parallels, any of the interactive applications built-in Varv could be expressed with OOP. However, there are a few key differences. Object-oriented code is imperative, which leaves less room for the underlying runtime to implement optimizations, and provides a more difficult target for higher-level tooling. Most object-oriented languages do not provide mechanisms for mod-
ification or extension of classes without changing the source code. In contrast, Varv concepts are declarative, inherently structured, and support modification via addition. Varv concepts consist of primitives, which enable the high-level yet expressive specification of application and interaction logic. Varv concepts do not provide mechanisms for encapsulation, such as private variables. The lack of encapsulation forces Varv applications to replicate the store design paradigm from Flux applications and aids rapid prototyping. Additionally, Varv concepts support modification and extension via addition, enabling new workflows for the development of interactive applications.

14.7.3.2 Feature-Oriented Software Development

Incremental [60] or Feature-Oriented Software Development (FOSD) is an area of research that provides mechanisms to incrementally develop software one feature at a time [6]. There are two general approaches to FOSD: compositional and annotative. Compositional approaches enable the development of features in distinct modules that can later be composed to create fully working applications. Most research implements compositional techniques as extensions to existing languages [5, 8, 23] but tools for adding compositional feature development to arbitrary languages exist as well [7, 23]. Annotative approaches enable feature-oriented development using explicit annotations of source code, such as #ifdef [147]. In general annotative approaches provide greater flexibility because they allow the modification of source code at the statement level, while compositional approaches provide better organization because code associated with each feature is modular and self contained [6]. Varv concepts implement a compositional approach to and retain similar limitations to past compositional systems. Compositional techniques generally do not provide mechanisms to introduce code fragments where order matters [147]. Within the context of Varv, this means there are certain cases where extending Varv programs requires duplication of existing code. Additionally, programs written using a compositional FOSD paradigm can be difficult to reason about because the final program results from multiple distinct artifacts [147]. Higher-level tools which visualize the final combined program can help [147, 148]. We believe similar tooling could be developed for Varv as well.

14.7.3.3 Conceptual Design of Software

Software engineers have long realized that they can build more complex and more efficient applications by sharing and reusing software components [217]. Déjà Vu [247] identifies that many web applications are built using combinations of similar components and provides a catalog of self contained and reusable components — called concepts — which can be integrated using a declarative template language to
build non trivial applications. Déjà Vu identifies that concept oriented architectures can allow for incremental development of applications by adding one concept at a time and testing functionality. In Déjà Vu the user is able to utilize concepts from a core catalog, and this catalog can be used to implement a wide variety of applications. However, if the user wants to implement their own concepts they need to write a frontend component and a backend server implementation.

Varv’s approach, including our choice to name its core building block a “concept,” is deeply inspired by Déjà Vu and Jackson’s writing on concept design [141, 142]. Varv implements a similar architecture in which concepts are bound to the UI using a template language. However Varv provides a lower level catalog of abstractions, such as actions which can be used for modifying state, and triggers which can be used for listening to state changes or user interactions. Varv focuses on providing declarative mechanisms for users to compose lower level abstractions into higher level semantic or domain specific abstractions. Varv also allows users to extend existing concepts and define new concepts without dropping into JavaScript. By providing mechanisms for extension, Varv allows incremental development one feature at a time.

14.8 Discussion

14.8.1 Limitations

Varv is a research prototype and, as such, it does not yet provide all the features necessary for building production-grade interactive software. There are two classes of missing features: those which are straightforward to implement but missing due to time constraints and those which require careful thought and are potential research questions for future work. In the first category are issues such as the lack of support for accessing remote or asynchronous data, the expressive limitations of Varv’s templates compared to templates found in popular frameworks such as React or Vue, and the relatively small standard library of actions and events provided by Varv. In the second category are issues such as the lack of access controls, the choice of template languages for alternative substrates, the inability to extend templates via addition, the challenges in authoring incrementally developed applications, and the challenges of supporting polymorphism in Varv. We expand on each of these issues from the second category below.

Information hiding and access controls. Concepts have no notion of private properties, which means any concept can access the properties of any other concept. This lack of information hiding is a conscious design choice because it replicates the Store design pattern—a common approach adopted by frontend libraries (e.g.,
14.8 Discussion

Redux, Vue, and Svelte) where application state is managed centrally to simplify developing cross-cutting interface elements. In doing so, Varv facilitates rapid prototyping and extension but this limitation makes it challenging to write interactive software that contains private secrets, such as API keys and passwords, or that relies on limiting read or write access to data to specific users, such as chat applications. Varv does offer a limited workaround: users can define local data stores that are not synchronized. These local stores allow users to store things like configurations but are not suitable for secrets since the data is still accessible by other concepts. It remains to be seen if we can augment Varv with a concise, descriptive, and legible syntax for annotating data with identity information and access controls while preserving the rapid prototyping affordances of our current approach.

**Definition files and templates for alternative substrates.** Varv is agnostic to the view layer, but the current template files and bindings rely on the existence of a declarative syntax (HTML) for representing the DOM. One of the design goals for Varv is to decouple application logic from interaction modality because we realized early on that it would be valuable to enable the rapid retargeting of interactions from one modality to another. We plan to integrate Varv with substrates outside the DOM environment, such as a WebGL view to support 3D or AR rendering or an IoT substrate that supports declarative interactive logic for intelligent devices such as lights and switches. We believe this is possible but are unaware of declarative template languages for expressing the view or, in the abstract, bindings between concepts and these substrates.

**Template modification via addition.** Varv’s templates support composition via template references, but when users add new features to Varv applications, previous templates and template refs often have to be copied and modified, complicating the development process, duplicating code, and effectively breaking with the open authorial principle [22]. The challenges of supporting template modification via addition may be a limit posed by compositional methods to extension. In compositional approaches to feature-oriented software development, it is considered impossible to introduce statements in the middle of existing methods [147]. If we consider the template definition synonymous with a function definition, this limitation is also applicable to templates. AspectJ provides a unique approach by enabling the extension of method calls within specific methods [155]. However, this multi-level approach to extension can be challenging to reason about and only covers certain cases, such as overriding a nested template in a specific parent template.
AUTHORING CHALLENGES. While the presented debugging tools are a first step to support authoring in Varv, the nature of accruing changes over time, possibly in many concept definitions and templates, poses new questions regarding how tools can best support authoring incrementally developed applications: If an application is edited by multiple users over longer periods of time and each modification is added through addition, users have to traverse each file, mentally tracking the incremental development of the application’s concepts, actions, and triggers, in order to understand the application state. Future tools might support users by making it possible to inspect the current state of an application behavior, i.e. presenting the user the merged concept definition and templates. While this would condense code into a single concept definition and template, such a process might lose information about how and in which order modifications were developed. Providing context and provenance for changes would be important.

SUPPORTING POLYMORPHISM. When Varv injects a source concept into a target concept using extension mechanisms, the target concept inherits the actions and properties of the source concept, but Varv does not form an is a relationship between the source and target concept. The target concept cannot populate properties or views which refer to the source concept, even though the target concept exposes the same interface of properties and actions as the source concept. This limitation is due to the event flow model. Allowing multiple concept types to appear in the same set of contexts could create a set of contexts with diverging states, for example, if an action is defined differently in the various concepts. Without polymorphism, it is not easy to create sets of similar objects which each have unique behavior.

EVALUATING USABILITY. We have evaluated the feasibility and expressivity of Varv through demonstration [179]. However, we have not evaluated the usability of the programming model with actual users. A user study of user interface systems such as our work with Varv is challenging [179, 241]. Currently, our tooling for development is proof-of-concept, and an—ideally longitudinal—study of software appropriation over time with Varv would require extensive tool support. Additionally, a user study would be required to understand whether users who are proficient programmers but unfamiliar with declarative programming, can make use of Varv.

We chose an event driven architecture because event architectures are well suited for incremental development [193]. Users can write new actions which run before or after any existing action or UI event in a Varv application, without changing the existing action or UI event. However, over-use of events can inhibit comprehensibility and debuggability of larger programs [193]. Our authoring and debugging
tools (see Section 14.6.1 and 14.6.2) let users edit templates and view data associated with concepts, but future work could explore richer visualizations or tracing and debugging of the event graph, potentially easing the developer experience in larger applications. Additionally, a new runtime could explore alternative programming styles such as event-driven functional reactive programming [273], or functional programming, which avoids issues of declaratively managing state.

14.8.2 Future Work

With Varv we have demonstrated that a declarative approach to specifying interactive applications as data structure is not only possible, but also provides a range of powerful capabilities. Through two very different applications built on top of Varv we have demonstrated that the ceiling for what can be achieved with Varv is high, but we do not clearly know its bounds in terms of expressivity and performance. Thus, an immediate opportunity for future work would be to more systematically evaluate these two aspects.

To better assess Varv’s performance, future work could begin by conducting comparative benchmark studies. Following the approaches used to evaluate the performance of frontend JavaScript libraries, these studies could measure both the performance (i.e., time taken) as well as memory consumption of running a suite of operations like rendering, manipulating, and updating thousands of interface elements. Besides empirical methods, future work on performance optimization can also look to practices already adopted by these frontend libraries as well as techniques detailed in the academic literature on dataflow management. For instance, as updating the DOM can be a computationally-intensive operation, React selectively updates DOM nodes by maintaining an in-memory virtual DOM [94]. Similarly, the data stream management community has developed methods for incrementally processing data by flagging data tuples as either new or removed, and only passing these flagged tuples (rather than the full data table) between dataflow nodes [1, 10].

Future work on determining Varv’s expressive ceiling can unfold in myriad ways. Our choice of implementing a todo list for our first case study was motivated by TodoMVC [290], which provides a benchmark to compare how various Model-View frameworks implement todo list applications. A next step would then be to target alternate benchmarks such as the seven challenging GUI programming tasks from 7GUIs [158]. To scale this approach, one could turn to large datasets of interactive applications [74] and interaction traces [73] to catalog common classes of interaction techniques, and decompose them into recurring conceptual design patterns — an approach that Déjà Vu has already begun to explore [247]. While promising, these directions adopt primarily qualitatively methods to determine expressivity. An
alternate approach might follow McGuffin and Fuhrman [214] to more formally evaluate Varv’s expressivity.

An exciting avenue for future work, and a direction inspired by the effect Vega [273] and Vega-Lite [272] have had in data visualization, would explore higher-level systems for authoring Varv applications. In particular, by representing interactive software as a data structure, Varv makes it possible to programatically reason about the composition of applications. As a result, one can imagine building not only freeform direct manipulation graphical authoring environments (akin to Lyra in the Vega/Vega-Lite ecosystem [271, 323]) but also methods for recommending and autocompleting interaction design (analogous to Data Voyager [313] or Juxxt [286]). For instance, a higher-level system might analyze the schema of concepts currently in use, and execute a lookup in the catalog to identify other concepts that are often used together or that have a complimentary schema. Besides the catalog described in the previous paragraph, such workflows would require the development of additional infrastructure to support a “concept ecosystem,” i.e., mechanisms to package and share concept definitions [126].

Such programmatic reasoning about the concepts that underlie interactive software also recalls ideas of instrumental interaction described by Beaudouin-Lafon [25]. Namely, Beaudouin-Lafon envisions a future where interaction techniques — reified [28] as “instruments” — rather than applications are the primary organizational unit of user interfaces. Thus, he imagines that interaction instruments can be reappropriated and used in contexts they were not initially designed for (e.g., using a snap-to-grid feature, typically found in vector graphics packages, but to organize the icons on your desktop). To realize such a vision, however, will require more sophisticated methods to compose and extend concepts than Varv currently supports. Here, one may look to the operators described by Jackson [142] such as action or structure (schema) synchronization, or Project Cambria’s [186, 187] approach of bidirectional lenses [107, 131, 132]. Reasoning about and applying the appropriate composition operators automatically would be a critical step on the journey to a cognitively convivial information space [108, 134].

14.9 CONCLUSION

Modern software development techniques for constructing interactive software typically involve writing imperative code which is packaged and deployed as hermetically-sealed turnkey applications. Writing extensible software is an explicit choice and requires careful design choices. In contrast, Varv provides a declarative approach to developing software, yielding an accretive development process and applications that are inherently extensible. We have outlined the design
goals of Varv and have explained how the components of the Varv language—*concepts, schema, actions*—help fulfill those design goals. We have demonstrated through two case studies the development process enabled by Varv, showing how Varv can be used to construct a domain specific toolkit for building board games, and how Varv can be used to collaboratively and incrementally develop a shared todo list feature by feature. We provide two examples of higher level tooling built on top of Varv, an inspector for accessing relevant code directly from an application’s UI and an alternative Blockly-based editor interface. We hope that Varv inspires future research to enable non-programmers to develop interactive software.

**ACKNOWLEDGMENTS**

We thank Geoffrey Litt, Daniel Jackson, Philip Tchernavskij, and the anonymous reviewers for their helpful feedback. We also thank Jonas Oxenbøll Petersen for assistance in preparing the video figure. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 740548) and from Carlsbergfondet (grant agreement No CF17-0643).

**APPENDIX**

The appendix of this paper is included as Appendix C.
Part III

APPENDIX
The appendix of Paper B, which was included as Chapter 10.

A.1 SCENARIOS AND DISRUPTIONS

(a) Scenario “Master’s Thesis.” Photo by Dato (WMAM), https://commons.wikimedia.org/wiki/File:Collaboration_with_Yerevan_State_University_of_Languages_and_Social_Sciences_after_V._Brusov_01.jpg. Licensed under CC 4.0: https://www.creativecommons.org/licenses/by-sa/4.0/deed.en. The original has been cropped. Icons made by Freepik from www.flaticon.com.

(b) Disruption for the “Master’s Thesis” scenario.

Figure A.1: Scenario A and associated disruption.

(b) Disruption for the “Book” scenario.

Figure A.2: Scenario B and associated disruption.
A scenario and disruptions

C

- PhD student and supervisor
- Jointly writing a scientific paper
- Co-located

DISRUPTION:

The paper gets rejected. Meanwhile, the supervisor has gotten a job abroad.

- The PhD student and the supervisor must now work together remotely.
- They want to resubmit the paper in time for the student’s thesis deadline.

(b) Disruption for the “Scientific Paper” scenario.

Figure A.3: Scenario C and associated disruption.
APPENDIX PAPER D: CODESTRATES V2

The appendix of Paper D, which was included as Chapter 12.

B.1 TODO LIST EXAMPLE CODE

B.1.1 View

```html
<div id="main" style="display: none;"> 
  <section class="hero is-dark"> 
    <div class="hero-body"> 
      <div class="container"> 
        <h1 class="title">TODOS</h1> 
      </div> 
    </div> 
  </section> 
  <section class="section"> 
    <div class="content"> 
      <input class="input is-rounded" id="newTodo" type="text" placeholder="New todo"> 
      <div id="todos" class="section"></div> 
    </div> 
  </section> 
</div>
```

B.1.2 Style

```css
body { 
  text-align: center; 
} 
body input { 
  width: 25em !important; 
  text-align: center; 
} 
.todo { 
  margin: 5px; 
  cursor: pointer; 
} 
.todo.completed { 
```
B.1.3 Controller

JavaScript Fragment

```javascript
// Load external CSS framework uploaded as an asset.
await wpml.requireExternal("bulma.css");

// Show main element when bulma has loaded
document.querySelector("#main").style.display = "";

// Load the code fragment containing our data (#model)
// Fragment.one(selector) works like querySelector and finds the first matching fragment
let model = Fragment.one("#model");

let todos = document.querySelector("#todos");
let input = document.querySelector("#newTodo");

input.addEventListener("keyup", (e) => {
  if (e.key !== "Enter") return;
  let newTodo = {
    text: input.value,
    completed: false
  };
  getData((data) => {
    data.todos.push(newTodo);
    storeData(data);
    input.value = "";
  });
});

data.getShowTodosFromData();

// Here we listen for changes to the model fragment
model.registerOnFragmentChangedHandler(() => {
  getData(showTodosFromData);
});

function showTodosFromData(data) {
  if (!data || !data.todos) return;
  todos.innerHTML = "";
  for (let todo of data.todos) {
    let todoDiv = document.createElement("div");
    let style;
    if (todo.completed) {
      style = "completed";
    } else {
      style = "is-info";
    }
    todoDiv.style = style;
    todos.appendChild(todoDiv);
  }
});
```

292
B.1 todo list example code

todoDiv.innerHTML = `<span class="todo tag ${style}
  is-large">${todo.text}<button class="delete
  is-small"></button></span>`;
todos.append(todoDiv);

let del = todoDiv.querySelector("button");
del.addEventListener("click", (e) => {
e.stopPropagation();
let todoDiv = e.target.closest("div");
deleteTodo(todoDiv);
});

todoDiv.addEventListener("click", (e) => {
let todoDiv = e.target.closest("div");
let index = getTodoIndex(todoDiv);
getData(data => {
data.todos[index].completed = !data.todos[index].completed;
storeData(data);
});
});

function deleteTodo(todoElement) {
let index = getTodoIndex(todoElement);
getData(data => {
data.todos.splice(index, 1);
storeData(data);
});
}

function getTodoIndex(todoElement) {
let index = Array.from(todoElement.parentElement.children).indexOf(todoElement);
return index;
}

function getData(callback) {
model.require().then(data => {
callback(data);
}).catch(err => {
console.log("Couldn't parse JSON data");
callback(undefined);
});
}

function storeData(data) {
model.raw = JSON.stringify(data, null, 2);
}
B.1.4 Model

```json
{
    "todos": [
        {
            "text": "Write some code",
            "completed": false
        },
        {
            "text": "Eat some food",
            "completed": false
        },
        {
            "text": "Sleep",
            "completed": false
        }
    ]
}
```
APPENDIX PAPER F: VARV

The appendix of Paper F, which was included as Chapter 14.

C.1 VARV LANGUAGE EXAMPLE

C.1.1 Abstract Concept Definition

```json
{
    "concepts": {
        "todo": {
            "schema": { "label": "string" }
        },
        "todoList": {
            "schema": {
                "todos": { "array": "todo" },
                "todosCount": { "number": {
                    "derive": {
                        "properties": [ "todos" ],
                        "transform": [
                            { "length": "todos" }
                        ]
                    }
                } }
            }
        },
        "actions": {
            "addNewTodo": [
                { "new": {
                    "concept": "todo",
                    "with": {
                        "label": "@newTodoLabel"
                    },
                    "as": "newTodo"
                },
                { "append": {
                    "property": "todoList.todos",
                    "item": "$newTodo"
                }}
            ]
        },
        "todoInput": {
            "schema": {
                "text": "string"
            }
        },
        "actions": {
```
Listing C.1: Example of an abstract concept definition of a todo list app consisting of three concepts.

C.1.2 Concrete Concept Definition

```json
{
  "concepts": {
    "todo": {
      "actions": {
        "deleteOnClick": {
          "when": {
            "click": {
              "view": "deleteButton"
            }
          },
          "then": "remove"
        }
      },
      "todoInput": {
        "actions": {
          "activateInput": {
            "when": [
              { "key": {
                "key": "Enter",
                "focus": "todoInput"
              }},
              { "click": {
                "view": "addTodoButton"
              } }
            ]
          }
        }
      }
    }
  }
}
```
C.1.3  Template

Listing C.2: Example of an concrete concept definition of a todo list app.

```xml
<dom-view-template>
  <template name="todo">
    <div>
      <span class="text">{text}</span>
      <span view="deleteButton">Delete</span>
    </div>
  </template>

  <h2>Todo List</h2>

  <div concept="todoInput">
    <h3>Add New Todos</h3>
    <input value="{text}" />
    <button view="addTodoButton">Add Todo</button>
  </div>

  <div concept="todoList">
    <h3>Todos ({todosCount})</h3>
    <div class="list">
      <div property="todos">
        <template-ref template-name="todo"/>
      </div>
    </div>
  </div>
</dom-view-template>
```

Listing C.3: Example of a template of a todo list app.

C.1.4  Custom Action

```javascript
/**
 * Usage: define a function "foo" in global scope.
 * { "customJS": { "func": "foo" } }
 */
class CustomJSAction extends Action {
  constructor(name, options) {
    if (typeof options === "string") {
      options = { func: options ;
    }
    super(name, options);
  }

  async apply(contexts, actionArguments) {
    if (this.options.func == null) {
      throw new Error("'func' must be set");
    }
  }
}
```
Listing C.4: Example action which lets users run arbitrary global functions as actions.

C.2 LIST OF PRIMITIVE ACTIONS

C.2.1 Concept Actions

<table>
<thead>
<tr>
<th>ACTION NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;count&quot;</td>
<td>Returns the count of instances of a given concept type. Filtering like in &quot;where&quot; is possible.</td>
</tr>
<tr>
<td>&quot;exists&quot;</td>
<td>Returns a Boolean variable of whether there exist instances of a given concept type. Filtering like in &quot;where&quot; is possible.</td>
</tr>
<tr>
<td>&quot;get&quot;</td>
<td>Returns the value of a property of either the current target or from another concept instance.</td>
</tr>
<tr>
<td>&quot;new&quot;</td>
<td>Creates a new instance of a given concept with the given properties. Has an option to not select the newly created instance.</td>
</tr>
<tr>
<td>&quot;remove&quot;</td>
<td>Removes the current target concept instance or instances stored in a variable.</td>
</tr>
<tr>
<td>&quot;set&quot;</td>
<td>Sets the value of a property or variable.</td>
</tr>
</tbody>
</table>
### C.2 list of primitive actions

#### C.2.2 Control Flow Actions

<table>
<thead>
<tr>
<th>ACTION NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;eval&quot;</td>
<td>Returns the Boolean value of a filtering expression.</td>
</tr>
<tr>
<td>&quot;exit&quot;</td>
<td>Terminates the action chain.</td>
</tr>
<tr>
<td>&quot;limit&quot;</td>
<td>Limits the number of context to a given count starting from the first or last.</td>
</tr>
<tr>
<td>&quot;run&quot;</td>
<td>Runs an action with a copy of the current event and then continues with the action chain independent of the outcome of the other action.</td>
</tr>
<tr>
<td>&quot;select&quot;</td>
<td>Selects all instances of a given concept type. Filtering like in &quot;where&quot; is possible.</td>
</tr>
<tr>
<td>&quot;storeSelection&quot;</td>
<td>Stores the current selected targets in a variable in the event.</td>
</tr>
<tr>
<td>&quot;switch&quot;</td>
<td>Tests conditions for several branches and executes the action chain of the branch that matches. A default branch can be set and it contains an option to continue after a successful branch.</td>
</tr>
<tr>
<td>&quot;wait&quot;</td>
<td>Stops and waits with continuing the action chain for a given duration.</td>
</tr>
<tr>
<td>&quot;where&quot;</td>
<td>Filters the current selection according to a given property or variable condition. Conditions can be combined using “and”, “or”, and “not.”</td>
</tr>
</tbody>
</table>

#### C.2.3 Boolean Actions

<table>
<thead>
<tr>
<th>ACTION NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;toggle&quot;</td>
<td>Inverts a Boolean property or variable.</td>
</tr>
</tbody>
</table>
### String Actions

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;concat&quot;</td>
<td>Concatenates an array of strings or variables to a new string.</td>
</tr>
<tr>
<td>&quot;enums&quot;</td>
<td>Returns an array of all possible enums of a string property.</td>
</tr>
<tr>
<td>&quot;length&quot;</td>
<td>Returns the length of a string.</td>
</tr>
<tr>
<td>&quot;textTransform&quot;</td>
<td>Transforms a string to uppercase, lowercase, or capitalization.</td>
</tr>
</tbody>
</table>

### Number Actions

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;calculate&quot;</td>
<td>Calculates a given mathematical expression.</td>
</tr>
<tr>
<td>&quot;decrement&quot;</td>
<td>Decrements a number property or variable by a given value.</td>
</tr>
<tr>
<td>&quot;increment&quot;</td>
<td>Increments a number property or variable by a given value.</td>
</tr>
<tr>
<td>&quot;random&quot;</td>
<td>Returns a random number within a given range.</td>
</tr>
</tbody>
</table>

### Array Actions

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;append&quot;</td>
<td>Appends an item to an array property or variable.</td>
</tr>
<tr>
<td>&quot;items&quot;</td>
<td>Returns the items of an array. Allows for filtering items.</td>
</tr>
<tr>
<td>&quot;length&quot;</td>
<td>Returns the length of an array.</td>
</tr>
<tr>
<td>&quot;prepend&quot;</td>
<td>Prepends an item to an array.</td>
</tr>
<tr>
<td>&quot;removeFirst&quot;</td>
<td>Removes the first item from an array.</td>
</tr>
<tr>
<td>&quot;removeItem&quot;</td>
<td>Removes the item on the given index from an array.</td>
</tr>
<tr>
<td>&quot;removeLast&quot;</td>
<td>Removes the last item from an array.</td>
</tr>
</tbody>
</table>
### C.3 List of Primitive Triggers

#### C.3.1 Reactive Triggers

<table>
<thead>
<tr>
<th>Trigger Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;action&quot;</td>
<td>Triggers when another given action is executed. Contains an option to specify whether it should trigger before or after the other action was executed.</td>
</tr>
<tr>
<td>&quot;interval&quot;</td>
<td>Triggers in intervals after a given time.</td>
</tr>
<tr>
<td>&quot;stateChanged&quot;</td>
<td>Triggers when a given concept or property changes.</td>
</tr>
</tbody>
</table>

#### C.3.2 View Triggers

<table>
<thead>
<tr>
<th>Trigger Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;click&quot;</td>
<td>Triggers when a given concept, property, or view is clicked.</td>
</tr>
<tr>
<td>&quot;key&quot;</td>
<td>Triggers if a given key is pressed and optionally a concept, property, or view is in focus.</td>
</tr>
<tr>
<td>&quot;mouseDown&quot;</td>
<td>Triggers when a given concept, property, or view receives a mouseDown event.</td>
</tr>
<tr>
<td>&quot;mouseMove&quot;</td>
<td>Triggers when a given concept, property, or view receives a mouseMove event.</td>
</tr>
<tr>
<td>&quot;mouseUp&quot;</td>
<td>Triggers when a given concept, property, or view receives a mouseUp event.</td>
</tr>
</tbody>
</table>

### C.4 List of Data Stores

<table>
<thead>
<tr>
<th>Data Store Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;dom&quot;</td>
<td>Stores data in the DOM of a website or webstrate. Listens to changes to these elements.</td>
</tr>
<tr>
<td>&quot;memory&quot;</td>
<td>Stores data in memory that gets emptied after a page refresh.</td>
</tr>
<tr>
<td>&quot;localStorage&quot;</td>
<td>Stores data in the localStorage.</td>
</tr>
</tbody>
</table>
### C.5 List of Extensions

<table>
<thead>
<tr>
<th>Extension Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;inject&quot;</td>
<td>Adds the schema and actions of one or multiple source concepts into a target concept. Properties or actions with the same name are overwritten by injected concepts in the order they are specified.</td>
</tr>
<tr>
<td>&quot;join&quot;</td>
<td>Combines one or multiple source concepts into a new concept. Properties and actions with the same name are handled like in the &quot;inject&quot; extension.</td>
</tr>
<tr>
<td>&quot;omit&quot;</td>
<td>Removes the given properties and actions from a target concept.</td>
</tr>
<tr>
<td>&quot;pick&quot;</td>
<td>Takes the given properties and concepts of a source concept and creates a new concept based on those.</td>
</tr>
</tbody>
</table>

### C.6 List of DOM View Template Tags

#### C.6.1 Template Tags

<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;dom-view-template&gt;</td>
<td>Indicates the start and end of a template in the DOM view.</td>
</tr>
<tr>
<td>&lt;template name=&quot;some-name&quot;&gt;</td>
<td>Allows to create named templates that can be reused using template references.</td>
</tr>
<tr>
<td>&lt;template-ref name=&quot;ref-name&quot;&gt;</td>
<td>Allows to reference a named template and insert it.</td>
</tr>
</tbody>
</table>
## C.6.2 Template Attributes

<table>
<thead>
<tr>
<th>ATTRIBUTE NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>concept</td>
<td>Indicates that the given element should be rendered for each instance of the given concept.</td>
</tr>
<tr>
<td>property</td>
<td>Indicates that the given element refers either to a property with type concept or to an array concept.</td>
</tr>
<tr>
<td>view</td>
<td>Indicates that the given element is a view that can be referred to in concept definition files.</td>
</tr>
<tr>
<td>value</td>
<td>Indicates that the given value of an text input, select or checkbox should be synchronized with the given property.</td>
</tr>
</tbody>
</table>


334


[318] Daisy Yoo, Aurélien Tabard, Alix Ducros, Peter Dalsgaard, Clemens Nylandsted Klokmose, Eva Eriksson, and Sofia Serholt. “Computational Alternatives Vignettes for Place- and


COLOPHON

This document was typeset using the typographical look-and-feel classicthesis developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst’s seminal book on typography “The Elements of Typographic Style.” classicthesis is available for \LaTeX: https://bitbucket.org/amiede/classicthesis/.

Published Version (September 30, 2022): Typos, grammar mistakes, and bibliography mistakes were corrected compared to the “Submitted Version” that the committee received on February 28, 2022.