Between Principle and Pragmatism: Reflections on Prototyping Computational Media with Webstrates

Marcel Borowski, Bjarke V Fog, Carla F Griggio, James R Eagan, Clemens N Klokmose

To cite this version:

Marcel Borowski, Bjarke V Fog, Carla F Griggio, James R Eagan, Clemens N Klokmose. Between Principle and Pragmatism: Reflections on Prototyping Computational Media with Webstrates. ACM Transactions on Computer-Human Interaction, 2022, 10.1145/3569895. hal-03860618

HAL Id: hal-03860618
https://telecom-paris.hal.science/hal-03860618

Submitted on 18 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Between Principle and Pragmatism: Reflections on Prototyping Computational Media with Webstrates

MARCEL BOROWSKI, Aarhus University, Denmark
BJARKE V. FOG, Aarhus University, Denmark
CARLA F. GRIGGIO, Aarhus University, Denmark
JAMES R. EAGAN, Télécom Paris, Institut Polytechnique de Paris, France
CLEMENS N. KLOKMOSE, Aarhus University, Denmark

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.

CCS Concepts: • **Human-centered computing → Interactive systems and tools**; *Empirical studies in HCI*; Web-based interaction; Collaborative interaction.

Additional Key Words and Phrases: computational media, malleable software, Webstrates, Codestrates, dynamic media, reconstructible media

ACM Reference Format:
Marcel Borowski, Bjarke V. Fog, Carla F. Griggio, James R. Eagan, and Clemens N. Klokmose. 2022. Between Principle and Pragmatism: Reflections on Prototyping Computational Media with Webstrates. *ACM Trans. Comput.-Hum. Interact.* Author Version, To Appear (October 2022), 50 pages. https://doi.org/10.1145/3569895

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. *Software as computational media* is a vision of an alternative model to the application-centric one. It blurs the distinction between who is the developer and who is the user of software and the boundaries between application domains [15]. 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

Authors' addresses: Marcel Borowski, marcel.borowski@cs.au.dk, Aarhus University, Department of Computer Science, Aarhus, 8200, Denmark; Bjarke V. Fog, bfog@cc.au.dk, Aarhus University, Department of Digital Design and Information Studies, Aarhus, 8200, Denmark; Carla F. Griggio, carla@cs.au.dk, Aarhus University, Department of Computer Science, Aarhus, 8200, Denmark; James R. Eagan, james.eagan@telecom-paris.fr, LTCI, Télécom Paris, Institut Polytechnique de Paris, 91120, Palaiseau, France; Clemens N. Klokmose, clemens@cs.au.dk, Aarhus University, Department of Computer Science, Aarhus, 8200, Denmark.

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Computer-Human Interaction*, https://doi.org/10.1145/3569895.

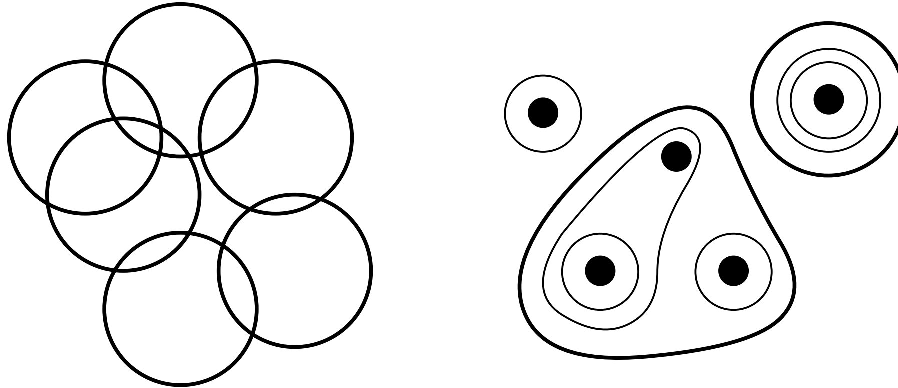


Fig. 1. Applications (left) versus computational media (right). diSessa [14] describes that "[m]onolithic, nonmodifiable applications have gaps and overlaps" (left) and, in contrast, a "computational medium allows seeding with small but extendible tools (black dots), but these tools can be organically enriched, altered, and combined, as successive layers here show" (right) (figure adapted from [14] and reprinted from [50] with authors' permission).

four activities can happen in the same medium or in different depending on how the *user* combines their tools. Figure 1 shows how diSessa illustrates the distinction between software as applications and as computational media. diSessa [14] regards functionality in applications as "nonmodifiable" while computational media enables extending and combining functionality.

In our work, we study the potential of computational media[1] as means for computer users to take control over their software and as a material foundation for the development of computational literacy.

We have articulated four principles that modern computational media should strive towards [35, 51]: That it should be *malleable*, so that users can adapt and repurpose their tools and documents in idiosyncratic ways. That it should be *computable*, so users can execute arbitrary computations in any document and in any use situation. A modern computational medium should 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, computational media 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.

We see computational media as collections of *information substrates* [4] — software artifacts that combine content, computation, and interaction and can be treated as documents or tools depending on their use. A particular information substrate can exhibit characteristics of a conventional application but can also be shared and repurposed based on the affordances of the computational medium.

Webstrates [35] 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 work: a webstrate (web + substrate) is a web page where any changes to its content, behavior, and appearance are automatically persisted on the Webstrates server and synchronized with all the users that are currently viewing it. Thus, any webstrate can be collaboratively

[1]We have previously used the term *Shareable Dynamic Media* [35] as a play on Kay and Goldberg's *Personal Dynamic Media* [33]. We later adopted the more straightforward *Computational Media* as introduced by diSessa and Abelson [15] for the same vision. We acknowledge that computational media is now also used in everyday language as an umbrella term for interactive software such as games, visualizations, interactive fiction, etc. In our work, we use the concept to denote software that exhibits a specific set of qualities that we will unfold in the following.

edited in real-time, including edits to program code embedded in the page. Klokmose et al. [35] demonstrated how Webstrates supports the creation of software that allows users not only to collaboratively author documents in real-time but also to collaboratively extend and reprogram their authoring tools while they are in use. Whether software served as a tool or as the object of a computing activity became a phenomenon of use instead of being dictated by the software itself. Over the years, Webstrates has been used to explore these software qualities in the contexts of design [12, 30], scientific work [50], public libraries [24, 66], video editing [36], programming assignments [9], data analytics [1, 27, 46], collaborative affinity diagramming [40], and video conferencing [25].

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 tensions between design principles and practical matters. 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 three examples of past Webstrates-based projects where some of these tensions have materialized. Furthermore, we introduce Codestrates v2 and Cauldron [5]: a software development platform and code-authoring tool based on Webstrates that instantiate the latest decisions and compromises in our exploration of the tensions between our principled vision and practical limitations. We report on the outcomes of a three-week programming challenge using Codestrates v2 and Cauldron, describing how this particular path in the realization of computational media affected users' understanding and appreciation of it. Last, we contribute six lessons learned from our attempts to practically realize a modern vision of computational media:

- 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 mechanisms to support switching between synchronous and asynchronous collaboration in programming are essential — particularly for collaborative programming.
- 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 2 positions our work towards related work on reprogrammable software and computational media. Section 3 summarizes the Webstrates and Codestrates platforms, and Section 4 presents a historical overview of their development. Section 5 describes eight tensions that have influenced the development of Webstrates and have been key concerns in the translation of vision to practice. Section 6 presents three Webstrates-based projects and examples of how tensions have materialized in them. Section 7, Section 8, and Section 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 10 discusses our lessons learned in the process of realizing the vision of computational media.

2 RELATED WORK

The vision of computational media originates from Kay's first idea of the Dynabook [33] — a personal computer to support children in "learning by doing." This idea evolved into a vision of personal computing that should be

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” [32].

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” [32]. 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 [15]. diSessa pointed out a 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” [13]. 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 [64].

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* [14] and created, together with Abelson, the Boxer platform [13, 15]. diSessa and Abelson [15] 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. [28] created Squeak based on the Smalltalk language which was used in the interim Dynabook prototype by Kay and Goldberg [32]. 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 [29]. 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 [45]), 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. [17] 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. [56], who implemented a system of modifiable buttons which “encapsulate appearance and behavior that is user tailorable” and that “are persistent objects and may store state relevant to the task they perform” [56]. MacLean et al. [44] 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" [42]. 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 [41]. Mackay also observed that users not only *adapt to* software (i.e., learn how to use it) but also *adapt software* to their own needs, either by customizing or appropriating (i.e., repurposing) it for tasks other than the ones it was designed for [43]. These findings opened the door to designing software that is powerful by not only allowing tailorability, but also by inspiring *appropriation* [16, 63], for example, by incorporating design qualities that support "unintended use" [57] or inviting users to add their own purposes to interface elements [11, 23].

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* [53] and so-called *no-code* or *low-code* tools. Literate computing environments [48] are software environments where editable and executable code is interleaved with text, images, and other rich media. No-code or low-code tools such as Notion,[2] Coda,[3] or AirTable[4] allow users to piece together personal software artifacts (e.g., spreadsheets, project timelines) or *computational media* to serve their own idiosyncratic needs and goals [21]. The literate computing paradigm has become widespread in the data science community with the popularization of Jupyter Notebook [54] and similar computational notebook software (see an overview in Lau et al. [39]). They are often used for iterative and exploratory data analysis [58]. Codestrates [59] 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 (Integrated Development Environments) [6].

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 [14] 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[5] and Pharo[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. [18] reflected on users' experiences with ad hoc interoperation of devices and services:

> *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*

[2]Notion: https://www.notion.so/ (Retrieved December 31, 2021)
[3]Code: https://coda.io/ (Retrieved December 31, 2021)
[4]AirTable: https://www.airtable.com/ (Retrieved December 31, 2021)
[5]GemStone: https://gemtalksystems.com/ (Retrieved December 31, 2021)
[6]Pharo: https://pharo.org/ (Retrieved December 31, 2021/)

> *potentially provides less application support than has been traditional. We believe that new UI techniques can help to maintain the usability that might be missing otherwise.* (Edwards et al. [18])

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 seeing 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 specific studies on how users understand computational media and how this understanding shapes the way they create software.

Finally, most related research 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 offers us ideas and principles that are still valid, our work is motivated just as much by asking ourselves how computational media might look today. With this paper, we want to contribute our experience on using modern web technologies for the realization of computational media, the tensions they create between principled design and pragmatism, and our latest observations with users.

3 BACKGROUND: WEBSTRATES AND CODESTRATES

In this section, we present Webstrates, and the two iterations of our development and authoring environment Codestrates v1 and v2. This section is based on previous work [5, 8, 35, 59] and describes *what* the platforms are. The next section will focus on the *why* and the motivations which drove our prototyping of computational media through building these platforms. A basic understanding of the technical background and challenges in realizing the platform is necessary to understand some of the tensions (see Section 5) and results from the game challenge (see Section 7), that are based on technical limitations and challenges that has led to pragmatic design decisions.

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 a webstrate.

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 a 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 [20] on the document (see details in [35]). 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

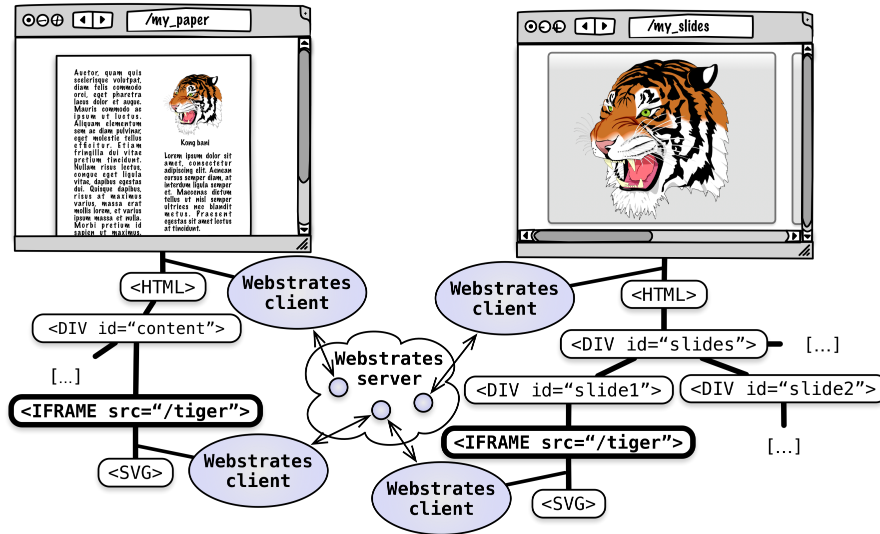


Fig. 2. Transclusion in Webstrates makes it possible to embed one webstrate into another by using the `<iframe>` element. This makes it possible to compose documents, e.g., a webstrate containing a SVG graphic of a tiger can be embedded inside a text document (left) and a slide show (right) at the same time. Changes to the graphic are immediately visible in the embedded versions as well (reprinted from [35] with authors' permission).

operations, e.g., an insertion of a character, to give a sense of real-time collaboration, e.g., seeing another user typing in a paragraph.

Webstrates can be composed using the hypermedia principle of *transclusion* [49]. Transclusion is an important aspect of the composability of documents. It strengthens both the malleability and shareability of webstrates. Figure 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. *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 adding read and write permissions on a per-webstrate level.

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 run-time, *shareability* by synchronizing persisted changes across clients, and basic *computability* using scripts in the DOM.

[7] APIs were added to Webstrates over time and also after its initial paper [35]. Transients, signaling, and permissions, for instance, were added later and first published in the Codestrates v1 paper [59].

3.2 Codestrates v1 Overview

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

Codestrates v1 [59] took inspiration from computational notebooks such as Jupyter [54] and follows the literate computing paradigm [53], 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 allows for creating application-like software such as the grocery list app shown in Figure 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 a so-called body paragraph containing the interface full screen, e.g., for a 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 elements and display them 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 user interface (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, prevents 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.

Codestrates v1 has a runtime environment for managing the execution of JavaScript in code paragraphs within a *codestrate*—a webstrate that runs Codestrates. Code paragraphs can be executed manually or set to *run-on-load*, which executes them whenever the codestrate is visited. 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

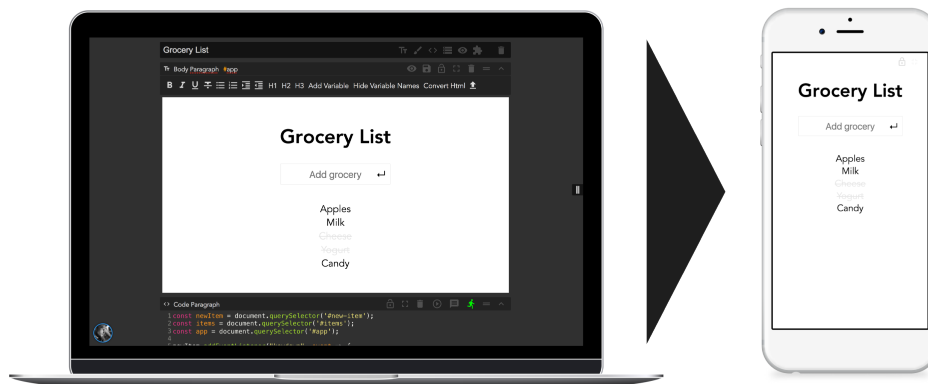


Fig. 3. A grocery list developed using Codestrates v1. Content and interactivity is authored in a notebook format. The main view of grocery list can be made full-screen and shown on a mobile device.

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 [8]. The package manager uses temporary transclusion to facilitate the moving of content between codestrates.

3.3 Codestrates v2 Overview

Codestrates v2 [5] 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 as 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 editable paragraphs in a computational notebook interface, Codestrates v2 introduces the concept of *code fragments*. Code fragments are reified pieces of code that can be programmatically 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. be executed a single time when a page is (re-)loaded — similar to the run-on-load of paragraphs in Codestrates v1. "Live reloading," which can be found in some JavaScript frameworks, reloads the web browser window after a file is changed in the IDE or code editor. In contrast, Codestrates v2's auto-run merely flags a code fragment to be *automatically run* when a webstrate is loaded — so that the it is not necessary to run it manually with the run button. Auto-run does not support liveness in the sense that changes to code are immediately applied.

Fragments with markup as well as fragments with style sheets can only be set to auto-run and cannot be run manually on demand. However, they are updated live and do not require a page refresh. Content from markup and style sheet fragments is rendered or preprocessed into 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. However, changes to the rendered views (e.g., from user interactions or edits via the browser's developer tools) are not transferred to the fragments (see Figure 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 5) rather than to a computational notebook. We did this to make the authoring environment more accessible to users unfamiliar with computational notebooks. When collaborating synchronously, Codestrates v2 provides basic awareness tools within Cauldron: The tree browser indicates when other users edit or view a fragment and inside the editor Codestrates v2 supports seeing the cursors and selections of other users who are currently editing a fragment.

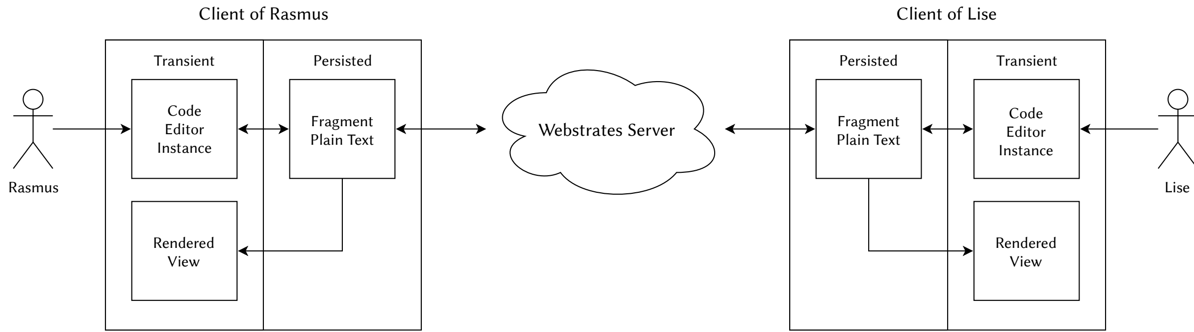


Fig. 4. Data synchronization of HTML fragments between two clients: Fragment plain text content is synchronized both ways to the Webstrates server and code editor instances. The rendered preview is also synchronized to fragment plain text changes, however, is not synchronized back to the fragment (reprinted from [5] with authors' permission).

Codestrates v2 also comes with the more generalized Webstrates Packages Manager. 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.

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 presents a timeline

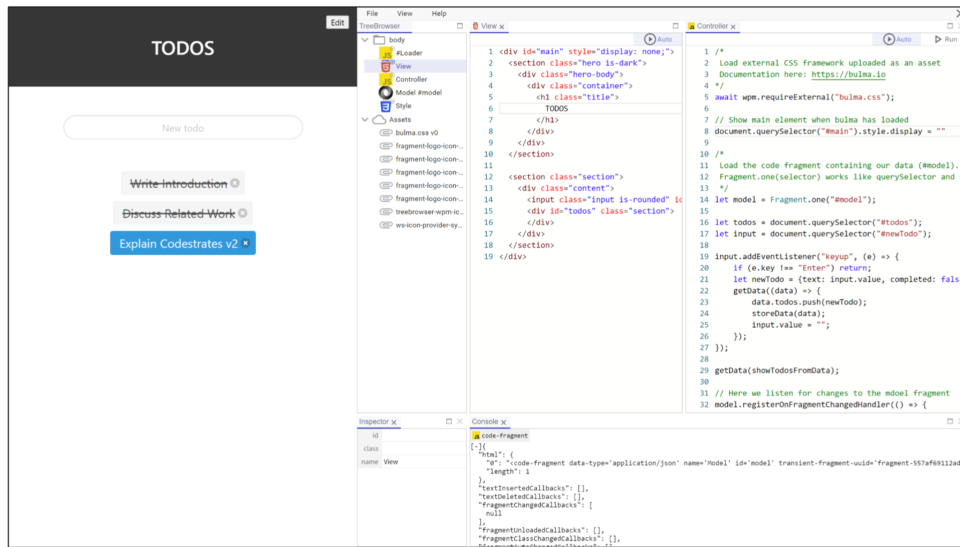


Fig. 5. A screenshot of a web page with Cauldron docked to the right side of the window. The "edit" button in the left half of the screen — the application — can be used to open Cauldron.

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 rationale of design decisions is important for the understanding the tensions in the next section. Figure 6 shows a visual timeline of important milestones.

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 [34] revisited the notions of application and document to propose a novel architecture for building interactive systems, based on instrumental interaction [3] and applied to an ad-hoc multi-surface environment. Instrumental interaction conceptualizes interaction as being explicitly mediated by instruments or tools rather than direct on the object of interest as in direct manipulation. Software built on this principle are dynamic collections of tools rather than static applications — not unlike diSessa's characteristic of computational media. Shared Substance [22], further, proposed data-oriented programming as a flexible approach to build software based on these principles for 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 and viable for interactive and functional demonstrations but not necessarily for day-to-day use. The Scotty system [17] aimed to address this limitation through the pragmatic approach of integrating instrumental interaction [3] concepts directly into the macOS Cocoa

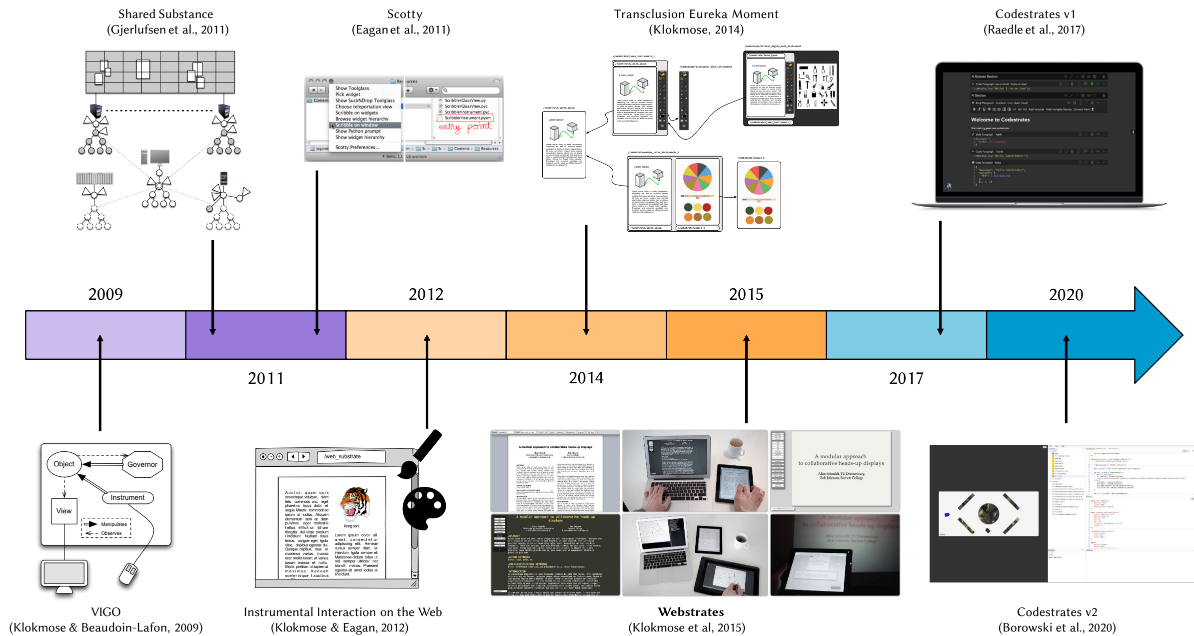


Fig. 6. Timeline of significant milestones in the development of Webstrates and Codestrates. To the left (purple) shows pre-Webstrates development leading to the formation of the vision in the middle (orange) and subsequent development of Codestrates to the right (blue).

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 principled purity of the research concepts and their resulting incongruity with the rest of the system.

Finally, with Webstrates [35], we made a compromise position between the two: it builds on existing web technologies and is able to draw on numerous existing web libraries, but bootstraps its own system. This approach yields a pragmatic 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.

4.2 Webstrates (2012–2016)

4.2.1 Forming the Vision (2012–2014). One of the realizations 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 [3] on the web inspired by the VIGO software architecture [34]. 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 extensions 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 [32] 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 [4].

4.2.2 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 manipulate the document of the other page as well as 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 was presented in the 2015 Webstrates paper [35]. 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.

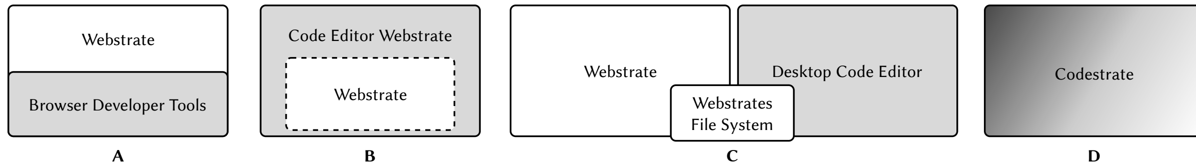


Fig. 7. The different relationships between a webstrate and tools for editing it in the chronological order they have been developed in. (A) A webstrate is edited directly through the developer tools in the browser. (B) A code editor webstrate is used to edit another webstrate through transclusion in a hidden `<iFrame>` element. (C) A webstrate is edited using a conventional desktop editor through the Webstrates File System bridge. (D) A webstrate as a codestrate where tools for editing it is embedded in it self.

In the beginning, all development happened through the developer tools of the web browser. From those it was possible to inspect the DOM of a web page and edit its content including embedded JavaScript and CSS style sheets. It was also possible to execute arbitrary JavaScript code in the web page 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" [35] that allowed for editing embedded JavaScript and CSS of another webstrate through transclusion (see Figure 7 for an overview of the different relationships between a webstrate and tools for editing it). 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.

4.2.3 Maturing of the Platform (2015–2016). After the first publication [35] 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,[8] to avoid overloading the server by many rapid changes to the DOM 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),[9] 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. This is a concrete example of a compromise between a principled pure approach in which all tools are built with Webstrates, and the pragmatic observation that letting users use their own tools would increase interest and adoption.

[8]D3.js: https://d3js.org/ (Retrieved December 31, 2021)
[9]Webstrates File System: https://github.com/Webstrates/file-system (Retrieved December 31, 2021)

4.3 Codestrates v1 (2016–2019)

4.3.1 Literate Computing with Codestrates v1 (2016–2017). In the 2010s, computational notebooks — the most popular being Jupyter [37] — introduced the idea of literate computing [53] and found acclamation in the field of explorative data science [58]. 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 [59] 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 (WYSIWIS) [61].

We, later, extended Codestrates v1 with a package manager, Codestrate Packages [8], 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 programing: 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* [15].

4.3.2 Codestrates v1 in Use (2018–2019). Over the years, Codestrates v1 transformed from a research prototype to a platform for creating applications with Webstrates. We and other colleagues used Codestrates v1 in a diverse range of projects: In Vistrates [1], 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 a limitation: 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 [9] (see Section 6.3 for more detail), 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 meant that, for instance, styles applied to an application, e.g., changing the background of all `<div>` elements, were also applied to the UI of Codestrates v1. In the same way, removing all `<div>` elements from the body, for example, to remove all items in a todo list, did also remove the implementation code 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 [9].

In another project together with a group of nanoscientists, we used Codestrates v1 to create a prototype of a *computational labbook* [50] (see Section 6.1 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, which was executing Perl and Python scripts that were sent from a codestrate using REST and WebSockets. Videostrates [36], 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[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. This is an example of a trade-off between principle and pragmatism: the need for integrating Webstrates with the world around it compromised the principle of everything in Webstrates being shared.

4.4 Codestrates v2 (2019–2020)

4.4.1 Decoupling and Generalization (2019–2020). Inspired by the 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 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 [5] consisted of the Webstrates Package Manager (WPM), the execution engine, and the Cauldron authoring environment.

4.4.2 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 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

[10]Monaco: https://microsoft.github.io/monaco-editor/ (Retrieved December 31, 2021)

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 which, for example, can be found in usual IDEs or a two-dimensional canvas where fragments can be freely moved around. Editors for fragments could be instantiated and placed anywhere in the user interface, and multiple editors could simultaneously show the same fragment. This means that it was, e.g., possible to put live editable code on a slide in a slideshow.

4.4.3 Separating the Authoring Environment (2020). While the execution engine provided helper functions to instantiate code editors to edit fragments, it did not create them by itself and instead required 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 took the pragmatic decision 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 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 a UI that is closer to what they might know from their experience with code editors and IDEs should make it easier to get started in using Codestrates v2.

Fragments are displayed in the editors in Cauldron by synchronizing the plain text content in the fragment with a code editor instance of a client. Changes to the plain text content of fragments, again, are persisted to the server. Thus, changes in code editor instances are persisted to the server and automatically update the code editor instances of other clients (see Figure 4).

In Codestrates v1, the authoring environment was — per default — always loaded in the form of the notebook interface. If a user built a full screen application with Codestrates v1 they could disable loading the authoring environment per default to speed up loading time.[11] In Codestrates v2, the authoring environment is *not* loaded per default. Instead, we provide an “Edit” button in the top right corner to load the Cauldron editing environment using the package manager and open it.

5 TENSIONS IN THE REALIZATION OF COMPUTATIONAL MEDIA

While revisiting and analyzing the development history of the Webstrates family of computational-media platforms, we identified eight tensions that had been driving our design decisions. By “tensions” we mean two opposing viewpoints that pull against each other, opening a spectrum in between them. These tensions reflect conflicts between the principles of our vision and the practical hurdles involved in creating a working system with limited resources at hand. For example, we envision software as a malleable medium, but in practice, the more malleable a system is, the harder it is to ensure its stability. Positioning our design decisions and trade-offs within such tensions allows us to characterize our explorations of computational media in a more structured way and systematically look for future alternatives. Next, we present an anecdotal description of the eight tensions we identified in our design decisions throughout the development of Webstrates and Codestrates v1 and v2:

- 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 shareable 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.

[11] To load it with the authoring environment enable one would then have to append `?edit` to the URL.

T4: *Big vs. Small Distance Between Development and Use Views* deals with how to enable the development of an application in the same context where it is used.
T5: *Authoring Environment Developed in Itself vs. Not Developed in Itself* similarly deals with whether the development 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.
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.

5.1 T1: Malleability vs. Stability

A key principle of computational media is to be malleable. Thus, software created with Webstrates should be able to be changed by users, allowing users to tailor their software in idiosyncratic ways. Malleability, however, does not only grant users the power to modify software 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, which are usually more stable as their functionality cannot be changed and always behave in exactly the same way.

Webstrates started as a very malleable, yet unstable, platform. Using primarily the web browser’s developer tools to modify a webstrate, everything within a webstrate could be changed. While this provided power, it also permitted users to easily break applications, e.g., by accidentally deleting the wrong elements from the DOM.

Codestrates v1 shifted the platform to more stability: Documents were 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, was still highly malleable and had a global scope, allowing users to modify existing parts of applications or adding new functionality. This global scope, on the other hand, also allowed 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 to boost stability. First, we exploited that it is possible to move content outside of the `<body>` element: the authoring environment, Cauldron, was not part of what is typically considered the content of a web page. This change added stability by preventing some user or third-party code from affecting Cauldron, for example, CSS styles that could accidentally break the layout of Cauldron. Second, the implementation code of Cauldron was loaded from another webstrate on page load and instantiated Cauldron within a transient element. This, again, made the platform more stable as the implementation code of Cauldron could not easily be corrupted. For example, if a user removed an element of the Cauldron UI accidentally, it would always be restored after a page refresh. On the other hand, these changes impaired malleability, as modifications of the Cauldron environment were not easily possible from within the document. Third and lastly, by employing the protected mode by default, Codestrates v2 prevented elements that were not explicitly approved by programmers — such as elements created by browser extensions or JavaScript libraries — from being persisted on the server and polluting the DOM, 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 are persisted equally.”

5.2 T2: What is Shared vs. What is Not Shared

There are different levels in which the state of an application can be shared. On the one end of the spectrum, everything is shared with other clients: UI elements, system state, and every visible change on the screen (e.g., the position of an image being dragged and dropped). On the other extreme, nothing is shared and users work in absolutely independent interfaces. Some degree of sharing is necessary to make a system collaborative, however, one may not want to share everything all the time, e.g., if a user opens a color picker to change the text color, they may expect the result to synchronize across clients without having the color picker appear for everyone.

The initial version of Webstrates shared the entire DOM by persisting it on the server. To have different views on documents, one needed to transclude "document-webstrates" into different types of "editor-webstrates." 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 to display UI elements without persisting them on the server. 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 [9] — similar problems can occur in collaborative writing tasks in online tools like Google Docs or Overleaf.

While Codestrates v1 shared elements by default and transient elements had to be explicitly indicated, Codestrates v2 moved towards *not* sharing elements by default, using the protected mode of Webstrates. This allowed users to interact with UI elements more freely without affecting the UI of others (e.g., if one opened a menu, the menu would stay closed in other clients) and also prevented the pollution of the persisted DOM with elements and properties injected by JavaScript libraries and browser extensions.

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 they perceive 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 process, such as editing an HTML file and reloading it to see the change. Of course, there is virtually always some level of indirection. Even in the example above, the developer tools present a mediated view of the 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. [2] 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 degrees of directness to modifying software. 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 — when 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 webstrates in two ways: (1) directly in the paragraph using the `contenteditable` attribute and (2) using a HTML editor built into Codestrates v1. 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 4). As a consequence, the user does not actually modify the DOM in the runtime, rather, the DOM is regenerated as 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 persisted and, thus, not synchronized across clients.

Approaches like React's[12] virtual DOM keep a virtual version of the DOM in memory. Whenever the virtual DOM is changed, React compares the new version of the virtual DOM with the old one, computes the changed elements, and only updates these changed elements in the real DOM to improve performance. Unlike this approach, applications in Codestrates v2 usually operate directly on the transient DOM generated from an HTML fragment. Rendering this transient DOM happens independent of the user's JavaScript code.

5.4 T4: Big vs. Small Distance Between Development and Use Views

The *development* and the *use* of applications are usually separate, e.g., developing them with an IDE and using them in a web browser. Our vision drives us towards blurring the line between development and use, however, many development tasks require specific tools (e.g., syntax check, debuggers), and these may need dedicated views that would be irrelevant or unwanted when just *using* the application. Different levels of separation or distance between views dedicated to *using* and views dedicated to *developing* software might affect how hard it is to do context switching between both activities. Switching from one context to the other can involve shifting focus from one part of the screen to another, from one application to another, or from one device to another (e.g., when testing mobile apps on a phone). We call this the distance between the development and use views of software, which affects the time and effort required to switch from one view to the other.

Webstrates allows for bringing the two views closer together and reduces this distance by using webstrates that inhabit both content, computation, and an authoring environment in the same document. Initially in Webstrates, editing happened either in the developer tools or a separate "editor-webstrate" that transcluded the webstrate to be edited (see Figure 7). 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 [53], Codestrates v1 weaved the development environment into the same space as the application. This avoided the need to switch between different views and narrowed the distance between development and use. Literate computing, however, is an atypical model for developing applications and therefore it could be sometimes difficult to mentally separate what is part of the development environment and

[12]React: https://reactjs.org/ (Retrieved December 31, 2021)

what is part of the application, e.g., confusing parts of the UI of Codestrates v1 with the UI of a word processor implemented in it [6].

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 between the UI of Codestrates v1 and the UI of the software created with it. In Codestrates v2, the Cauldron environment is separated from the application — like a browser's 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, however, still allows creating software where the distance is as low as in Codestrates v1, e.g., the simple Jupyter-notebook clone Ganymede [5].

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 [28] or Pharo. Such an authoring environment allows users to change core interactive 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 [1], the computational labbook [50], or a collaborative writing editor [6].

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 [31], 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.

5.6 T6: 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 (T5): 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 the code of the authoring environment was 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 in a persisted way into a webstrate.

5.7 T7: Creating Something New vs. Offering Something Familiar

The vision of computational media introduces a new way of thinking about software, which requires new concepts, tools, and infrastructure. To realize this vision, we could attempt a "pure," principled implementation from scratch, but this would prevent us from leveraging existing solutions such as mainstream programming languages, libraries and other resources, and users would face a steep learning curve. On the other hand, we can build on existing technologies (such as the web) to allow us *and* users to benefit from prior knowledge of familiar tools, programming languages and knowledge-exchange resources (e.g., Stack Overflow[13]). However, these were built with different underlying assumptions about how software works (e.g., that end-users would not change functionality on runtime), which impose limitations to how close we can get to our vision.

By building on existing web technologies, Webstrates allows developers to reuse existing programming knowledge and a vast selection of JavaScript libraries. This introduced problems when the libraries were designed under different assumptions than those of Webstrates. For example, some libraries manipulate the DOM when they are loaded under the assumption that the changes they make are ephemeral and reset on next page load, which is not the case with Webstrates. Developers had to rethink some of their development practices — especially if they developed for the "normal" web before — as Webstrates meddles with some of the fundamentals of web development such as creating persisted web application without providing an explicit back-end.[14]

Initially, authoring would happen through the developer tools of the browser, which were familiar to most with web programming experience, yet atypical in how changes made through the development tools were made persistent. The editor webstrate we built provided familiar IDE-like interface to programming, but introduced confusion around why it could not be used to edit the HTML of a webstrate but only embedded scripts and styles. Furthermore, it introduced confusion that editing scripts would require a reload of the webstrate to take effect but editing styles would not.

In Codestrates v1, most of the library issues from Webstrates persisted and were even amplified by interference with the authoring environment in the DOM: When using the common Bootstrap[15] 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. Users would suddenly have to reflect about how otherwise well-known libraries would effect their authoring environment. The introduction of transient elements, however, made it possible to manually adapt libraries to Codestrates v1, e.g., by rendering the output of visualization libraries such as Vega-Lite[16] 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

[13]Stack Overflow: https://stackoverflow.com/ (Retrieved December 31, 2021)
[14]While the Webstrates server is technically the back-end, development of applications in Webstrate usually only happens in the front-end.
[15]Bootstrap: https://getbootstrap.com/ (Retrieved December 31, 2021)
[16]Vega-Lite: https://vega.github.io/vega-lite/ (Retrieved December 31, 2021)

introduction of the run-on-load feature of paragraphs caused confusion, e.g., when users were not sure how to execute something after the page was loaded [6].

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 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 tree browser for code fragments — allowing users to reuse their experience with other code editors or IDEs. With Cauldron, we could provide the user with a tree-like browser to inspect and organize code fragments like files. But unlike a conventional IDE, the "files" were stored as DOM nodes and the "folders" were <div> elements in the DOM. For simple programs, the user would not need to be aware of this distinction, but for more complicated software with interdependence between fragments it would suddenly matter.

5.8 T8: Working With Your Own Tools vs. Adopting Built-in Tools

To modify a malleable medium, users require authoring tools. This can be either a user's own tool that connects to the medium, like a code editor or IDE, or built-in tools. 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. Additionally, a dedicated authoring environment can offer medium-specific features such as *run* or *auto-run* buttons or awareness features such as shared cursors for collaborative coding. To use conventional tools, a conventional medium such as a file-system would have to be emulated, and medium-specific features would be ignored.

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, pushing users towards using 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 prevents users from using, e.g., collaborative coding features such as shared cursors. Still, for some users, as, e.g., seen in [6], 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.

5.9 Discussion of the Tensions

The tensions fall out of the overarching trade-off of what we refer to as principle vs. pragmatism. 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.

In T2 (What is Shared vs. What is Not Shared), the tension 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.

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 ran 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 necessary to help users unlearn what they have learned or otherwise break with those prior assumptions.

We emphasize that the list of tensions is not exhaustive for all computational media and there could be other tensions at stake in other contexts. One of these might relate to scalability and size, e.g., "small-scale vs. large-scale projects." In our work, we have focused on the development of small-scale personal software for individuals or smaller groups — in line with early visions of computational media. The tensions that we have outlined are the ones that influenced design decisions during the development process of Webstrates and Codestrates. We discuss implications for computational media beyond the Webstrates family in Section 10.7.

6 TENSIONS IN PAST WEBSTRATES-BASED PROJECTS

To give a sense of how we have used Webstrates in our work, we will present three projects in which Webstrates has been the technical foundation. We will exemplify how some of the above tensions have materialized in the projects.

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 [50] using Webstrates and Codestrates v1. Through a series of participatory design activities over the course of one year, 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 computational labbook, they could create a computational workflow by drag-and-dropping various tools from a tool panel, and mix it with written notes. Figure 8 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 labbook aggregated a number of scripts and tools the scientists otherwise would use through the command-line and multiple applications. Each labbook was a self-contained computational environment that they could configure for a particular experiment, share with colleagues through its URL, and interact with

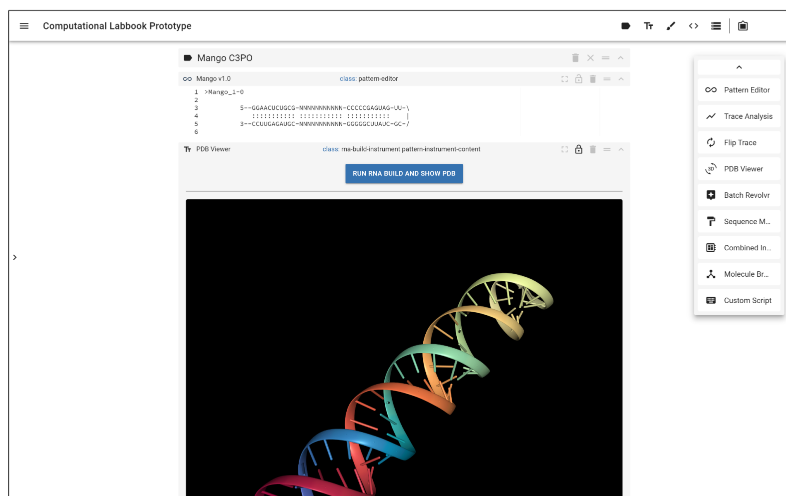


Fig. 8. The computational labbook prototype supports the computational design of RNA molecules. On the right side is a set of tools that can be dragged into the main work area in the center. For instance, the screenshot shows a tool that converts the ASCII representation of molecules into a 3D visualization (reprinted from [50] with authors' permission).

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.

In the computational labbook, stability was provided with a set of tools, or building blocks, that could be used plug-and-play to create new computational workflows — in line with the term *reconstruction* [15]. However, the scripts which were embedded in the tools are reprogrammable from within the labbook. In day to day work, the notebook could be used as a regular piece of software, but in the critical situation where reprogramming was needed, it was possible (T1: Malleability vs. Stability). 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 did not propagate when they were done to, e.g., accommodate a particular experiment, was meaningful, if they, however, were done to fix a general bug such as a memory leak, it was 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.

6.2 Building Prototypes for Public Libraries

In the PLACED project [24, 65, 66], 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 often 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 as a platform for rapid, exploratory, and iterative prototyping of new tools for both visitors to the libraries and organizers of events.

We built a prototype, PARTICIPATE, in which events were represented with what we called an *activity sheet* (see Figure 9). Activity sheets were inspired by event pages on social media such as on Facebook. Participants could share posts, images, and access books in the library related to the event. Organizers could 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 back-end development. This is not unlike how HyperCard was used in the 1980s to create functional prototypes in early participatory design projects [19].

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 and, 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

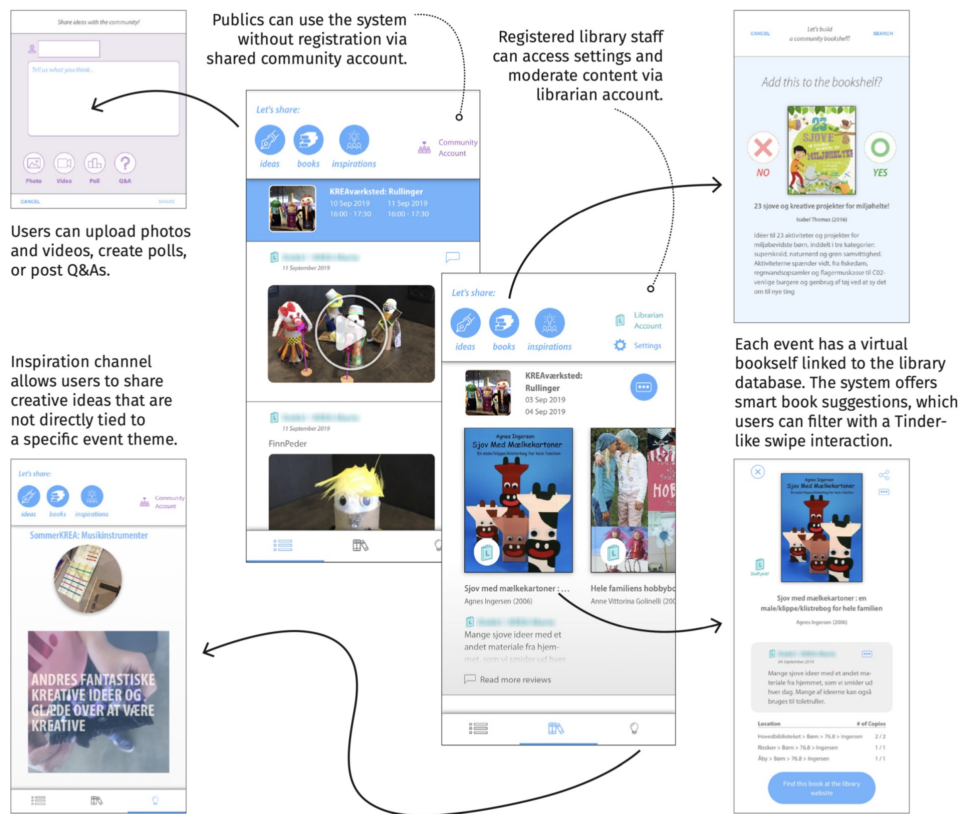


Fig. 9. The PARTICIPATE prototype. A high fidelity prototype of a web app built on top of Webstrates. It shares characteristics with a event page on social media where participants can post messages and pictures as well as share inspiration and references to books in the library. Instead of using a conventional server with a database, content is stored directly in a webstrate (reprinted from [65] with authors' permission).

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 realize fine grained access control. Currently, a user logged into a Webstrates server can have read or write access to a webstrate such as an activity sheet 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 relates to 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.

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 [9]. 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 [8]: (1) Teaching assistants (TAs) would prepare programming assignments in a private codestrate and push assignments as packages to an assignment repository. (2) Students would then pull the assignments to a codestrate that they collaborated on in pairs. (3) After students solved the assignment, they handed it in by pushing it to a submission repository. (4) TAs would then pull the last submitted version before the assignment deadline — enabled by the timestamped version history of a webstrate — into their own codestrate and grade it. (5) TAs, lastly, would push the graded assignment back into the submission repository of each respective group.

In another occasion, Codestrates v2 has been used to develop a tool to create interactive exercise sheets for programming (see Figure 10). 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: Malleability vs. Stability). We did not want the students to be able to break their exercise sheet unintentionally — a problem that happened while using the less stable Codestrates v1 in the interactive systems class. 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

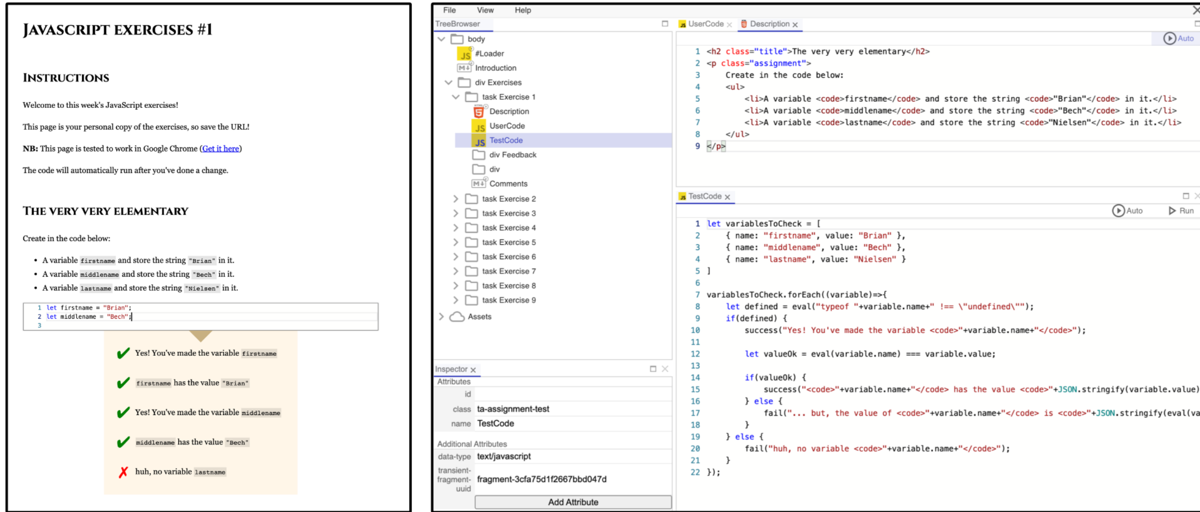


Fig. 10. Codestrates v2 based exercise sheet for teaching programming. Left shows the students' view of the exercise, right shows how the instructor can edit the exercise using Cauldron.

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 could cause the script to 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 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.

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 insights on how the particular instantiation of computational media realized through Codestrates v2 and Cauldron was experienced by users. 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 a 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. This is described in more detail in Section 7.2.

Participants	P1	P2	P4	P5	P6	P7	P8	P9	P10
Programming	8	2	8	8	6	8	7	8	10
Web Development	6	3	7	5	6	7	8	6	5
JavaScript	8	2	9	6	6	7	9	7	8
Game Development	5	4	5	1	3	1	2	2	6
Webstrates	8	3	7	7	2	9	4	5	1
Codestrates v1	4	3	5	6	2	1	2	5	3
Developed with Webstrates	yes	no	yes	yes	no	yes	no	no	no
Developed with Codestrates v1	no	no	yes	no	no	no	no	yes	yes

Table 1. Overview of the participants' self-assessed programming knowledge on a scale from 1 (no knowledge) to 10 (expert), and whether they developed with Webstrates and Codestrates v1 before. (Only participants that filled out the demographic questionnaire are listed.)

7.1 Participants

We recruited participants for the study creating posts promoting the study in the Slack workspace of Webstrates and on 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 or with Webstrates were not a requirement. 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 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 1. All questions were phrased as "knowledge of [skill]" on a scale from 1 (no knowledge) to 10 (expert) with no intermediate descriptors apart from numbers. Further, participants were asked if they had previously "developed with Webstrates/Codestrates v1" as yes-or-no questions. Some participants worked together in groups, Table 2 summarizes the groups and the games they implemented.

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.[17]

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.

[17] A grant agreement requires us to work with subjects on a volunteer basis.

Groups	G1	G2	G3	G4	G5	G6	G7	G8
Participants	P1	P2	P3	P4, P11, P12	P5	P6, P8	P7	P9, P10
Game	*Uruburu*	-	Card Game	*The Teachable Game*	*Flappy Bird*	Tank Game	-	*Rouge*

Table 2. Overview of the groups that worked together and the titles of the games they implemented.

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 no external editors. Participants were, however, allowed to use any JavaScript libraries, such as Phaser,[18] P5.js,[19] or Three.js.[20]

The documentation of the platform included the documentation of Webstrates, Codestrates v2, and Cauldron. We created a "getting started" video and for each of the three platforms an introductory documentation and a full API reference. 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 8). The examples were intended as a starting point for participants to see how applications and games could be implemented in Cauldron.

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 5 days, resulting in a total of 21 days for participants to work on the game challenge.

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.

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 who 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.

[18]Phaser: https://phaser.io/ (Retrieved December 31, 2021)
[19]P5.js: https://p5js.org/ (Retrieved December 31, 2021)
[20]Three.js: https://threejs.org/ (Retrieved December 31, 2021)

7.3 Data Collection

In the beginning of the game challenge, we collected demographic data about the participants (the introductory questionnaire can be found in the supplementary material of this article). 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.

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. Some interviews were conducted in Danish, those were then translated into English, striving for *equivalence* in meaning and interpretation in favor of transliteration [55]. 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.*"

The main goal of this analysis was to gather empirical insights about how the design decisions behind Codestrates v2 and Cauldron affected participants' experiences and understanding of computational media. For this purpose, we used the eight design tensions described in Section 5 for driving the analysis of the interviews and reflecting on how the design trade-offs that led to Codestrates v2 and Cauldron ended up helping or obstructing the creation of computational media. Thus, the results are not intended as an exhaustive picture of how participants used Codestrates v2 and Cauldron but as a discussion of how participants' breakdowns, frustrations, and confusions help us reconsider our design decisions and inspire new trade-offs when navigating the tensions between the vision and the pragmatics of computational media.

We followed a two-stage reflective thematic analysis [10] approach. In the first stage, we used the eight tensions in Section 5 as the guiding themes for a deductive coding of all interview transcripts. 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 in a spreadsheet, as well as annotating how the design trade-offs behind Codestrates v2 and Cauldron shaped users' experience and understanding of computational media (e.g., to what extent the tools met participants' expectations of malleability and stability).

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 identify nuances in the data and create subthemes). 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. As part of these discussions, we often re-watched the recorded demos and inspected the implementation code of the games to aid in our interpretation of the interview data. 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 [47]. 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.

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. We refer to participants as P1 – P10 and to groups as G1 – G8 in this and the following sections. G2 and G7 did not manage to implement a game, but we still report on their experience.

8.1 Group 1: Uruburu

P1 worked alone on his game *Uruburu* (see Figure 11a). The game of P1 is inspired by the video game Snake[21]: 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 puzzles.

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 by implementing the levels, design of the snake, and some basic gameplay mechanics such as moving the snake or adding walls. One head could be controlled by using the WASD keys, the other with the arrow keys. Further gameplay mechanics, like growing, eating items, or multiple levels, however, were not yet implemented.

8.2 Group 2

P2 tried to use Codestrates v2 and Cauldron alone but did not succeed due to lacking programming knowledge (see also Table 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.

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.

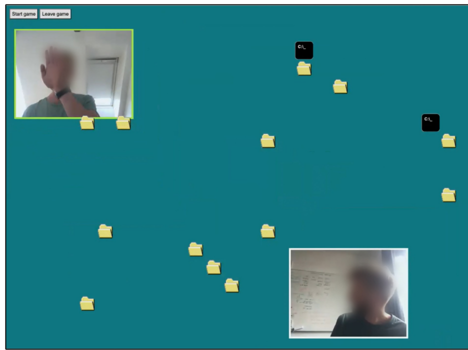
G3 developed a card game based on Uno[22] that would accommodate up to four players (see Figure 11b). In contrast with the original Uno game, this game was meant to allow players to dynamically alter the rules along the way. Further, the rules were envisioned to be arbitrary and not limited to a particular 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.

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.

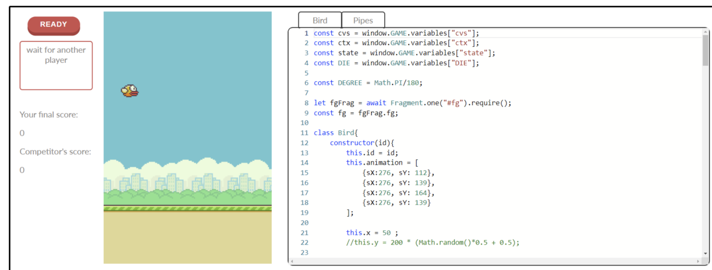
[21]Snake video game: https://en.wikipedia.org/wiki/Snake_(video_game_genre) (Retrieved December 31, 2021)
[22]Uno card game: https://en.wikipedia.org/wiki/Uno_(card_game) (Retrieved December 31, 2021)

(a) Group 1: *Uruburu*.

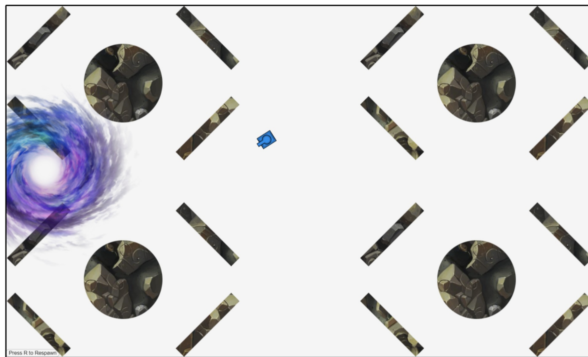
(b) Group 3: Card Game.



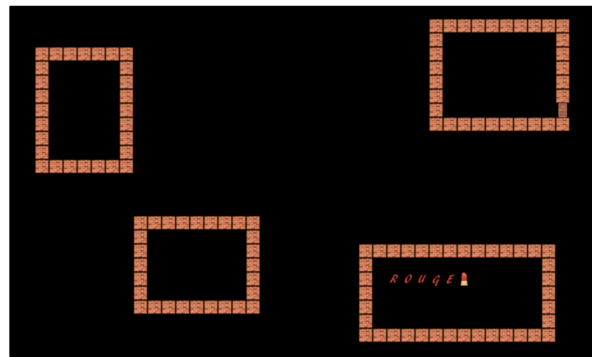
(c) Group 4: *The Teachable Game*.



(d) Group 5: *Flappy Bird*.



(e) Group 6: Tank Game.



(f) Group 8: *Rouge*.

Fig. 11. Screenshots of the games implemented by the participants.

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[23] 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 (see Figure 11c). 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 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 though they are immediately affected by them. The game was not fully finished as the bomb element was never written. Thus, it was possible to play the game but there were no conditions for winning or losing.

8.5 Group 5: Flappy Bird

P5 worked alone in creating a Flappy Bird[24] clone for multiple players (see Figure 11d). 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. To make the rules editable, P5 separated some of the game logic in a single code fragment that they then exposed to the player as part of the game window. In essence, the game window, thus, consists of both the actual game and the program code in which a player can manipulate parameters, e.g., for the gravity or the 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.

8.6 Group 6: Tank Game

P6 and and P8 worked together to create a 2D tank game (see Figure 11e). Unlike the other games that came out of the challenge, the tank game is a direct adaptation of an example[25] 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. G6 also considered implementing a scoring system, a leaderboard, and other game modes such as team deathmatch but, due to time constraints, did not.

[23]Hot Potato game: https://en.wikipedia.org/wiki/Hot_potato (Retrieved December 31, 2021)
[24]Flappy Bird: https://en.wikipedia.org/wiki/Flappy_Bird (Retrieved December 31, 2021)
[25]Codestrates Tank Trouble example: https://demo.webstrates.net/TankCauldron/release/?copy (Retrieved December 31, 2021)

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[26] in which the goal is to construct elaborate "Rube Goldberg-esque" contraptions. P7 explained 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.

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 1×1 space in the game field (see Figure 11f). 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 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.[27] 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.

9 RESULTS

We structure the results around the tensions from Section 5 to report on the aspects of the design of Webstrates, Codestrates v2, and Cauldron that impacted participants the most — especially focusing on the breakdowns, frustrations, and confusions they experienced during the challenge. We present findings on six out of the eight tensions, because we did not identify new insights for tensions T5 (Authoring Environment Written in Itself vs. Not Written in Itself) and T6 (Self-contained vs. Auto-updating Authoring Environment) in the data. For each of the six tensions, we present subthemes describing salient patterns 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 illustrative of open challenges for future work on computational media. Each section is concluded with a short summary of takeaways from the findings.

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, and in the possibility to break the development environment.

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 participants perceived it as a *real* versioning system that could be trusted. Some explained this was due to feeling a lack of control in what changes comprised a version and that this was "*not transparent*" (P10).

[26]The Incredible Machine: https://en.wikipedia.org/wiki/The_Incredible_Machine_(series) (Retrieved December 31, 2021)
[27]Baba Is You: https://en.wikipedia.org/wiki/Baba_Is_You (Retrieved December 31, 2021)

> *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 as 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 revision 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 [in the interface]. 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 cloning capabilities of Webstrates to recreate a working version of a broken system. For example, the tank game group (G6) had started their project by cloning 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['s camera feed] 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 [versions]. 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 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.

9.1.2 Losing Access to Cauldron. A few participants introduced bugs in the code of their games that caused them to lose access to Cauldron, which they would have needed for fixing the bugs. Since the game and Cauldron ran within the same browser tab, sharing state, screen real-estate and memory resources, some bugs in the game could also break the authoring environment itself. For example, P7 introduced a bug that would prevent the page from loading 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.

9.1.3 Hot Spots vs. Frozen Spots. Participants made explicit efforts and code-structuring decisions to control what parts of the games players should or should not change. A common approach to scoping the malleability of the game relied on defining *hot spots* and *frozen spots* in the game code, i.e., code that may be changed by players and code that should stay untouched. Participants seemed to expect most code to be *frozen*, except for the rules and mechanics that players were allowed to reprogram. For instance, P1 wanted to let players change the maze logic of his Uruburu snake game (e.g., letting the snake pass through a wall or not), so they created dedicated fragments for two types of hot spots: an HTML fragment called "game-area" with the HTML of the maze, and a JavaScript fragment called "uruburu-logic," where a function associated CSS classes of game elements to a Boolean determining whether the snake could go through them or not. In this way, players could see what CSS

class to assign a maze element to let the snake go through it, or change the logic for all elements tagged with a specific CSS class.

Some participants tried to make hot spots accessible to players by exposing code and selected parameters in the game interface itself. 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*" that affected the Flappy bird (see Figure 11d). P3 offered players a list of text boxes for redefining the behavior of each card in his game, where they could link the name of a card (e.g., "3-hearts") to how many cards it would make the player draw from the deck and whether it changed who the next player was.

We found it interesting that beyond helping players identify hot spots, most participants seemed to feel responsible for ensuring the stability of the game *themselves*. Even though the goal of the challenge was to build a reprogrammable game, there were concerns about Cauldron granting players uncontrolled access to the whole source code: "*if they can open the editor and change the way things bounce around, well they can change anything*" (P7). We believe that the more restrictive interfaces for changing the rules of the games, such as P3's textboxes or G4's dropdown menu, are not only attempts to make the rule-changing easier to use but also strategies to nudge restrictions into what users could change. For example, P5 feared that by having players change the code directly within the game rather than by using Cauldron they were more likely to introduce syntax errors, for which they imagined creating safeguards with block-based programming to "*make it simpler for non-programmers to understand what's going on.*"

9.1.4 Takeaway T1: Malleability vs. Stability. To ensure that users confidently can change malleable software, they need to be able to trust a versioning system and easily be 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 *hot spots* of the code to users while keeping the rest as *frozen spots*. Here, future computational media should provide explicit mechanisms to support creators in making these safeguards in a flexible manner. Finally, designers of computational media should carefully consider how to let users revert when breaking their own authoring environment.

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.

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 that they talked about fragments, e.g., "*those sort of fragments, they are not synchronized between clients*" (P4) — fragments *are* synchronized between clients; their generated DOM *view* is not.

Participants that had previous experience with Webstrates and Codestrates v1 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 the assumption that DOM elements would be automatically synchronized across 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)

9.2.2 Expecting Fragments to Share State Across Clients. A common surprise among participants was to realize that the automatic sharing of a webstrate'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,[28] a JavaScript framework that keeps views synchronized with their models, and then move the code to Codestrates v2 to automatically synchronize all players every time one of them changed a Vue.js model (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 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 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 9.5 elaborates on this confusion about how things were meant to be done.

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.

[28] Vue.js: https://vuejs.org/ (Retrieved December 31, 2021)

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 autogenerated pipes. They used signaling to synchronize the position of the birds when players made them jump. However, it was only after testing and seeing different game layouts across clients that P5 realized that the position of the pipes also had to be synced.

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)

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 at the time of execution. Live sharing of fragment changes across clients enables collaborative coding on Cauldron, however, the 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.

9.2.5 Takeaway T2: What is Shared vs. What is Not Shared. We learned that there are many nuances to what is expected to be shared or not in computational media. Participants' frustrations around system elements that were not automatically shared across clients suggest that their mental model of computational media is that *everything* is shared, and the difficulties that come with syncing changes across clients are also magically solved. Synchronizing the DOM of a website does not resolve issues around shared JavaScript runtime state. The state of, e.g., a `<canvas>` is not reflected in the DOM. Users of computational media need to be made aware of these nuances to adjust their expectations about what is shared and what is not.

9.3 T3: Editing Directly vs. Editing Indirectly

Participants struggled making sense of the divergence [2] between the representation of software when it is running (i.e, the JavaScript runtime), and the representation of the form in which it is being edited (i.e., the code fragments). The assumptions of Cauldron allowing direct editing of the running system led some participants to expect Codestrates v2 to support live programming environment, which caused breakdowns related to running the same code multiple time and dealing with unexpected side effects.

9.3.1 Lack of Awareness of How Much the DOM and Code Fragments Diverge. Code fragments hold the "dead" version of a piece of code, similar to an HTML or JavaScript file. However, participants often expected code fragments to represent the DOM, i.e., the "live," running system. Especially in the case of HTML fragments, many were surprised when realizing, through trial and error, that they could not directly edit the DOM view of fragments with Cauldron. For example, P3 complained, explaining "*one thing is the DOM and the other thing is the HTML fragment, and these two entities are different, it's not the same.*"

Some participants failed to realize the full vision of their rule-changing games because their designs were based on the assumption that Cauldron would let them access and edit the runtime version of their games. For example, P1 wanted users to customize the map of their snake-like game and share the resulting HTML code with others. Based on their experience with Webstrates, they expected users to customize the map by manipulating the DOM, i.e., right-clicking on tiles, inspecting them with the developer tools and altering their behavior by changing a CSS class (e.g., turning a "wall" tile into a "floor" tile). However, they soon realized that the changes done via the developer tools were not reflected on the map's HTML fragment, so sharing the code of custom maps in this way would not work: "*I had the full assumption that when you make an HTML fragment, that is the same as making DOM in sort of basic Webstrates*" (P1). P5's Flappy Bird exposed the JavaScript fragments of the "Bird" class directly on the game's interface (see Figure 11d). P5 expected users to edit values in the class definition (e.g., the gravity property) while they played, so that the birds would immediately change their behavior. This suggests that P5 thought they could change the behavior of the running bird instances by editing the code of the JavaScript fragment. However, when testing, they realized that editing the JavaScript fragment simply redefined the bird class, and that only future bird instances would behave differently.

9.3.2 Confusing Reprogrammability for Live Programming. Many participants seemed to expect Codestrates v2 and Cauldron to support *live programming*, even though the documentation does not suggest this is possible. We believe that the confusion might come from the fact that, unlike JavaScript fragments, static fragments such as HTML, CSS or Markup are re-rendered automatically after changes and do not feature a *run* button. We speculate that most often, users create HTML and CSS fragments first, getting a live-programming feeling from the start. This might also be the case for many JavaScript fragments that leave no evident side-effects after running them manually, e.g., fragments that only define classes, functions or global variables, which simply get overwritten after re-running the fragment. Eventually, unexpected side effects from previously ran JavaScript fragments affect the behavior of the system, and the impression of a live programming environment breaks.

For example, many participants perceived common precautions typical of JavaScript programming as shortcomings of Codestrates v2, such as checking if a DOM element exists before creating it or keeping track of event listeners to avoid duplications. P10 was surprised that pressing *run* on a JavaScript fragment would multiply programmatically-created views: "*And when we then pressed 'run,' it just like spawned a new window down further down in the game.*" P12 suggested a "solution" to avoid refreshing the page before running a JavaScript fragment:

> *It would be awesome if you could just say "run" and then say "unrun" and then make an edit and then "run." So you don't need this refresh, because it's something that takes quite a lot of time, and you are thrown a little out of context and such. [...] I mean, when you say "run," you execute a script. And if you then press "run" again, well then it runs the same code on top of it, and that messes it up.* (P12)

Echoing Basman et al. [2], event listeners were "a primary source of divergence," for which P1 proposed a solution:

> *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)

Such ideas for "solutions" suggest that these participants expected Codestrates v2 and Cauldron to support live programming and, even though they were typing JavaScript, expected the runtime environment to work differently. Future design iterations should either avoid expectations of liveness and help users keep track of how state is affected by running JavaScript fragments (e.g., by supporting declarative registration of event listeners [2]), or try to indeed support live programming in JavaScript, for example, by decoupling rendering from event handling [60].

9.3.3 Takeaway T3: Editing Directly vs. Editing Indirectly. Cauldron automatically updates HTML and CSS fragments for convenience, but this may have interfered in how participants made sense of the divergence between diverse types of code fragments (e.g., HTML vs. JavaScript) and their "live versions." Many participants assumed that editing and re-running code fragments would directly affect the running game (e.g., by changing existing DOM rather than re-rendering HTML), which also gave a false sense of working in a live programming environment. Future implementations of computational media should explore ways of mitigating confusions about how (in)directly users can edit the running system, highlighting the limitations of the medium (e.g., that code cannot be "un-run") to adjust users' expectations, or fully realizing a live programming environment.

9.4 T4: Big vs. Small Distance Between Development and Use Views

We found that the short distance between the Cauldron (the *development* view) and participants' games (the *use* view) supported fast context switches as we intended. For example, P4 liked that one can "*just go in and see 'okay, how is is implemented?'*" and P12 that "*if there then is something that suddenly doesn't work, you can go in and inspect what it is, [...] you have your development environment in the same place. It works well for this kind of prototyping-in-the-wild-ish thing.*" However, this short distance also caused problems related to shared resources between the views and extra effort in setting up Cauldron after every page refresh.

9.4.1 Shared Resources Between Development and Use Environment. 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)

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 click out of Cauldron, it would write into the text file [fragment] that you were working on, the W-A-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)

9.4.2 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, increasing the distance between the views himself:

> *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.

9.4.3 Takeaway T4: Big vs. Small Distance Between Development and Use Views. Keeping a short distance between the development and use views of malleable enable quick context switches but 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. By reducing the distance and running both the game code and the development environment in the same web page, they shared the same resources, and a slowdown of the game could cause Cauldron to be unresponsive. As the Cauldron is running the same web page as the software being built, reloading the game by refreshing the website also caused Cauldron to be reloaded. Future computational media should provide mechanisms to support both *in-use* as well as *out-of-use* development when needed. The latter to, e.g., avoid the development tools and application code fighting for the same resources.

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, which created confusions. Participants, for example, had difficulties matching the concept of a fragment with their existing concept of a file. Similarly, they expected Codestrates v2 to work in different ways depending on their prior experience with Webstrates and Codestrates v1.

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 participants compared fragments to 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 related confusions in Section 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)

This data suggests that small conceptual differences can cause breakdowns by instilling assumptions and expectations of how the system works. We believe that implementations of computational media that draw from existing tools and concepts (e.g., JavaScript, files) should clearly indicate that what feels familiar can also *differ* from its typical use (e.g., JavaScript fragments may look like files, but unlike files, they can be executed multiple times independent of page load).

9.5.2 Confusions Based on Prior Knowledge of Webstrates and Codestrates v1. Participants expected Codestrates v2 to work in different ways depending on their prior knowledge of Webstrates, Codestrates v1, or web development in general. 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. Traditionally, anything you put in the DOM in Webstrates would be persisted on the server unless were explicit about it being transient. In Codestrates v2, HTML fragments provided a convenient way of editing HTML. However, only the plain text content is persisted, and the rendering of it to the DOM is transient, although automatically updated when the HTML is edited. This caused confusion leading participants to assume that you could not just store data in the DOM in the conventional "Webstrates way." You could, but not by using HTML fragments, you would have to write directly to the DOM with the conventional JavaScript DOM API. This, for example, confused P4, who was familiar with Codestrates v1 and made use of the synchronized DOM in the past:

> *Well, those sort of fragments, they are not synchronized between clients. And that confused me a little because that was actually the way a lot, I mean, that was actually some of what I had taken advantage of in the old Codestrates [v1]. That, 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)

P1 even felt that these new features in Codestrates v2 diverged from the original vision of Webstrates:

> *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)

Similarly, G4 was insecure about how to store state in their game and thought they could not just store it directly in the DOM as they would normally do in Webstrates. In the documentation of Codestrates v2, we had given an example of how state could be stored as plain text in a JSON fragment, mimicking a traditional model-view-controller pattern, and they assumed that was the "correct" Codestrates v2 approach. However, in practice that led to a number of performance and concurrency issues in their implementation that would not had happened had they stored the data as DOM nodes.

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 portraying fragments as files and DOM elements as folders in the tree browser, participants built both helpful and confusing assumptions about how these work. Thinking of fragments as files helped some participants to organize their code, but also obscured differences about how they are executed (e.g., files are executed on page load, fragments can be executed multiple times with the "run" button), producing unwanted side effects in the registration of event listeners and programmatic creation of views. Participants with experience on Webstrates and Codestrates v1 found it frustrating not to be able to do some things in the same way as before, especially related to synchronizing state across clients. To prevent fundamental misunderstandings, the documentation should include careful explanations of examples with clarifications of the system differs from previous versions and traditional web programming.

9.6 T8: Working With Your Own Tools vs. Adopting Built-in Tools

We found that participants had mixed feelings about using the built-in authoring environment Cauldron, and were reluctant to abandon their personalized tools and known workflows for developing software. However, they also appreciated the benefit of launching Cauldron with a single click without any setup.

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 System (WFS) 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 [file] and write the script there. Okay, I can do that in Cauldron but I can do it in a text editor I like. And I need to import this CSS library. Is Codestrates helping me in any way to do that? No, actually it might be a little more difficult than the document[ation] 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 then, that's the reason for me, like, I started it [the game] 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 traditionally. 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.

9.6.2 Using One's Own Tools is Not Practically Possible. The WFS 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 tools like syntax highlighting, refactoring, and autocompletion in the editor. Codestrates v2's unconventional approach to storing code in custom fragments requires tailoring a tool like WFS specifically for that and at the time of the study this had not been done.[29]

That meant that at the time of the study it was not practically possible to use own tools for working on Codestrates v2 but participants had to use Cauldron. This tension is, thus, just as much a reflection of what participants might want in the future — or remember from working in earlier versions of 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)

The "*bad trade-off*" that P1 mentions could be addressed by developing better built-in tools or better possibilities to use own tools — or ideally both. For Cauldron, this could mean to improve the authoring environment with features like, e.g., a global search, refactoring, or debugging capabilities. For the ability to use own tools, an improved version of WFS with proper support for editing fragments and uploading assets would be a solution. Potentially provide an API for integrating awareness features for collaborative editing that, e.g., could be used in an extension for a conventional IDE such as Visual Studio Code.

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 without the requirement for additional tools, which is especially useful when only small edits need to be done. For more involved development work, experienced programmers — like most of our participants — only rarely wanted to abandon their familiar tools. A system should ideally strive to do both: offering an integrated authoring environment but allowing users to continue using their own tools — at least one, however, should be done well: a powerful integrated tool or a good integration into existing tools.

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

[29]Later one of the authors adapted WFS to work with Codestrates v2's fragments

individual 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 experiences that stood out in this particular case and had an impact on our current and future research.

10.1 L1: Reprogrammability Is Not the Same as Malleability

Our experiences with Webstrates have made it clear to us us that making software malleable to a user goes beyond merely making it reprogrammable. This is a point also Tchernavskij et al. [62] make 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. Even to a trained programmer, reprogrammable software written by others can be intractable, hard to reason about, and daunting to change. Mechanisms to reason about the composition and interactive behavior of software is necessary to gain the confidence to change it. Also, programmers have to be provided with mechanisms to experiment without the fear permanently damaging the software. These mechanisms have to be reliable and transparent. This is something that, for example, the Webstrates versioning mechanism was not to our participants. Hence, malleability is a relation between the technical mechanisms for changing software, and the user's perception of the software and the mechanisms to change it.

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 [38], hence, in the spirit of computational media, the mechanisms ideally should be similar for editing code and editing content.

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.

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 and across multiple clients or users. This insight motivated our work on a new declarative programming model, *Varv* [7], suited for computational media, that, we believe, represents the next step in our exploration of computational media. If creators of authoring tools for web-based computational media wish to leverage people's existing programming language skills, mediating software such as WebAssembly allows for the use of other programming languages in the browser. This does not, however, guarantee that users' mental understanding of the system is congruent with the state of the system. We would argue that this mental burden must be borne by the developer of the medium, and employing a declarative programming language largely removes the need for end-user programmers to play state machine in their heads.

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 middle ground 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.

We believe it is important to make users aware of the differences between the principled vision and a pragmatic implementation of it when introducing them. When focusing too much on the vision users can align their expectations with it and become frustrated or confused when the implemented platform differs from that, e.g., due to technical limitations.

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.

10.7 Discussion of the Lessons Learned

The lessons came from our own past and present experiences with computational media in the form of the Webstrates family of platforms. We do believe, however, that these lessons are of value to other types of computational media such as computational notebooks.

Computational notebooks are but one particular type of computational media, but by far the most popular kind. While different computational notebook environments have similarities, e.g., the overall system metaphor and structure, there are of course also differences: For instance, the programming model issues identified in L4 are also inherent in the Jupyter [54] notebooks, where each cell is run individually and it can be difficult to understand from which version a result is coming from [26], while the reactive model employed in Observable [52] alleviates some of these issues. Although still based on JavaScript, this reactive model at least ensures consistency in state between collaborators and between textual code and program state. It does not seemingly provide support for L2 as there are no mechanisms in place to separate private and communal editing of code and content, even if inconsistencies in state are solved. Interestingly, with regards to L5 and L6, the notebook format seems to have largely become an established concept that constrains users' expectations and ensures some level of familiarity. The risk is, then, that new forms of computational media might inadvertently make users draw upon the computational notebook as the basis for their understanding even when not meaningful or productive. Even for existing environments this is a risk. For example, just as we saw how Cauldron manifested itself as an "IDE-but-not-quite" to participants, Observable might also be seen as "Jupyter-but-not-quite" due to its different code execution model.

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 principled 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 results 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

We thank the participants of the game challenge and all co-authors and collaborators on the presented Webstrates projects. 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).

REFERENCES

[1] Sriram Karthik Badam, Andreas Mathisen, Roman Rädle, Clemens N. Klokmose, and Niklas Elmqvist. 2018. Vistrates: A Component Model for Ubiquitous Analytics. *IEEE Transactions on Visualization and Computer Graphics* (2018), 586–596. https://doi.org/10.1109/TVCG.2018.2865144

[2] Antranig Basman, Luke Church, Clemens Nylandsted Klokmose, and Colin Clark. 2016. Software and How it Lives On - Embedding Live Programs in the World Around Them. In *PPIG 2016 - 27th Annual Workshop*. https://www.ppig.org/papers/2016-ppig-27th-basman1/

[3] Michel Beaudouin-Lafon. 2000. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In *Proceedings of the 18th International Conference on Human Factors in Computing Systems (CHI '00)*. ACM. https://doi.org/10.1145/332040.332473

[4] Michel Beaudouin-Lafon. 2017. Towards Unified Principles of Interaction. In *Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter*. https://doi.org/10.1145/3125571.3125602

[5] Marcel Borowski, Janus Bager Kristensen, Rolf Bagge, and Clemens N. Klokmose. 2021. *Codestrates v2: A Development Platform for Webstrates*. Technical Report. Aarhus University. https://pure.au.dk/portal/en/publications/codestrates-v2-a-development-platform-for-webstrates(66e1d4d9-27da-4f6b-85b3-19b0993caf22).html

[6] Marcel Borowski and Ida Larsen-Ledet. 2021. Lessons Learned From Using Reprogrammable Prototypes With End-User Developers. In *End-User Development (IS-EUD '21)*. Springer. https://doi.org/10.1007/978-3-030-79840-6_9

[7] Marcel Borowski, Luke Murray, Rolf Bagge, Janus Bager Kristensen, Arvind Satyanarayan, and Clemens N. Klokmose. 2022. Varv: Reprogrammable Interactive Software as a Declarative Data Structure. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. ACM. https://doi.org/10.1145/3491102.3502064

[8] Marcel Borowski, Roman Rädle, and Clemens Nylandsted Klokmose. 2018. Codestrate Packages: An Alternative to "One-Size-Fits-All" Software. In *Proceedings of the 2018 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '18)*. ACM. https://doi.org/10.1145/3170427.3188563

[9] Marcel Borowski, Johannes Zagermann, Clemens N. Klokmose, Harald Reiterer, and Roman Rädle. 2020. Exploring the Benefits and Barriers of Using Computational Notebooks for Collaborative Programming Assignments. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*. ACM. https://doi.org/10.1145/3328778.3366887

[10] Virginia Braun and Victoria Clarke. 2006. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101.

[11] Marianela Ciolfi Felice, Sarah Fdili Alaoui, and Wendy E. Mackay. 2018. Knotation: Exploring and Documenting Choreographic Processes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM. https://doi.org/10.1145/3173574.3174022

[12] Peter Dalsgaard, Kim Halskov, and Clemens Nylandsted Klokmose. 2020. Chapter 7 - A study of a digital sticky note design environment. In *Sticky Creativity*. Academic Press, 155–174. https://doi.org/10.1016/B978-0-12-816566-9.00007-0

[13] Andrea A. diSessa. 1982. *A Principled Design for an Integrated Computational Environment*. Technical Report. MIT. http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TM-223.pdf

[14] Andrea A. diSessa. 2001. *Changing Minds: Computers, Learning, and Literacy*. MIT Press. https://mitpress.mit.edu/books/changing-minds

[15] Andrea A. diSessa and Harold Abelson. 1986. Boxer: A Reconstructible Computational Medium. *Commun. ACM* 29, 9 (1986), 859–868. https://doi.org/10.1145/6592.6595

[16] Alan Dix. 2007. Designing for Appropriation. In *Proceedings of HCI 2007 The 21st British HCI Group Annual Conference University of Lancaster (BCS-HCI '07)*. BCS Learning & Development Ltd., 27–30. https://dl.acm.org/doi/abs/10.5555/1531407.1531415

[17] James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, 225–234. https://doi.org/10.1145/2047196.2047226

[18] W. Keith Edwards, Mark W. Newman, Jana Z. Sedivy, and Trevor F. Smith. 2009. Experiences with Recombinant Computing: Exploring Ad Hoc Interoperability in Evolving Digital Networks. *ACM Transactions on Computer-Human Interaction* 16, 1 (2009). https://doi.org/10.1145/1502800.1502803

[19] Pelle Ehn and Morten Kyng. 1992. Cardboard Computers: Mocking-it-up or Hands-on the Future. In *Design at Work: Cooperative Design of Computer Systems*. 169–196.

[20] Clarence A. Ellis and Simon J. Gibbs. 1989. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. 399–407. https://doi.org/10.1145/67544.66963

[21] Bjarke Vognstrup Fog and Clemens Nylandsted Klokmose. 2019. Mapping the Landscape of Literate Computing. In *PPIG 2019 - 30th Annual Workshop*. https://www.ppig.org/papers/2019-ppig-30th-fog/

[22] Tony Gjerlufsen, Clemens Nylandsted Klokmose, James Eagan, Clément Pillias, and Michel Beaudouin-Lafon. 2011. Shared Substance: Developing Flexible Multi-Surface Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. https://doi.org/10.1145/1978942.1979446

[23] Carla F. Griggio, Arissa J. Sato, Wendy E. Mackay, and Koji Yatani. 2021. Mediating Intimacy with DearBoard: A Co-Customizable Keyboard for Everyday Messaging. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. ACM. https://doi.org/10.1145/3411764.3445757

[24] Carla Gröschel, Peter Dalsgaard, Clemens N. Klokmose, Henrik Korsgaard, Eva Eriksson, Raphaëlle Bats, Aurélien Tabard, Alix Ducros, and Sofia E. Serholt. 2018. PARTICIPATE: Capturing Knowledge in Public Library Activities. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems (CHI EA '18)*. ACM. https://doi.org/10.1145/3170427.3188605

[25] Jens Emil Grønbæk, Banu Saatçi, Carla F. Griggio, and Clemens Nylandsted Klokmose. 2021. MirrorBlender: Supporting Hybrid Meetings with a Malleable Video-Conferencing System. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. ACM. https://doi.org/10.1145/3411764.3445698

[26] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM. https://doi.org/10.1145/3290605.3300500

[27] Tom Horak, Andreas Mathisen, Clemens N. Klokmose, Raimund Dachselt, and Niklas Elmqvist. 2019. Vistribute: Distributing Interactive Visualizations in Dynamic Multi-Device Setups. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM. https://doi.org/10.1145/3290605.3300846

[28] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. https://doi.org/10.1145/263698.263754

[29] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. 2008. The Lively Kernel A Self-supporting System on a Web Page. In *Self-Sustaining Systems*. https://doi.org/10.1007/978-3-540-89275-5_2

[30] Mads Møller Jensen, Roman Rädle, Clemens N. Klokmose, and Susanne Bodker. 2018. Remediating a Design Tool: Implications of Digitizing Sticky Notes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM. https://doi.org/10.1145/3173574.3173798

[31] Josh Justice. 2021. Modifiable Software Systems: Smalltalk and HyperCard. (2021). https://2021.splashcon.org/details/live-2021-papers/9/Modifiable-Software-Systems-Smalltalk-and-HyperCard The Seventh Workshop on Live Programming (LIVE 2021).

[32] Alan Kay and Adele Goldberg. 1977. Personal Dynamic Media. *Comput. J.* 10, 3 (1977), 31–41. https://doi.org/10.1109/c-m.1977.217672

[33] Alan C. Kay. 1972. A Personal Computer for Children of All Ages. In *Proceedings of the ACM Annual Conference - Volume 1 (ACM '72)*. ACM. https://dl.acm.org/doi/abs/10.1145/800193.1971922

[34] Clemens Nylandsted Klokmose and Michel Beaudouin-Lafon. 2009. VIGO: Instrumental Interaction in Multi-Surface Environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. 869–878. https://doi.org/10.1145/1518701.1518833

[35] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. ACM, 280–290. https://doi.org/10.1145/2807442.2807446

[36] Clemens Nylandsted Klokmose, Christian Remy, Janus Bager Kristensen, Rolf Bagge, Michel Beaudouin-Lafon, and Wendy Mackay. 2019. Videostrates: Collaborative, Distributed and Programmable Video Manipulation. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. ACM, 233–247. https://doi.org/10.1145/3332165.3347912

[37] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks—a publishing format for reproducible computational workflows. *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (2016). https://doi.org/10.3233/978-1-61499-649-1-87

[38] Ida Larsen-Ledet and Henrik Korsgaard. 2019. Territorial Functioning in Collaborative Writing. *Computer Supported Cooperative Work (CSCW)* (2019), 391–433. https://doi.org/10.1007/s10606-019-09359-8

[39] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. 2020. The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. https://doi.org/10.1109/VL/HCC50065.2020.9127201

[40] Jiali Liu and James Eagan. 2021. ADQDA: A Cross-Device Affinity Diagramming Tool for Fluid and Holistic Qualitative Data Analysis. *Proceedings of the ACM on Human-Computer Interaction* 5, ISS, Article 489 (2021). https://doi.org/10.1145/3488534

[41] Wendy E. Mackay. 1990. Patterns of Sharing Customizable Software. In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work (CSCW '90)*. ACM, 209–221. https://doi.org/10.1145/99332.99356

[42] Wendy E. Mackay. 1991. Triggers and Barriers to Customizing Software. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '91)*. ACM, 153–160. https://doi.org/10.1145/108844.108867

[43] Wendy E. Mackay. 2000. Responding to Cognitive Overload: Co-adaptation Between Users and Technology. *Intellectica* 30, 1 (2000), 177–193.

[44] Allan MacLean, Kathleen Carter, Lennart Lövstrand, and Thomas Moran. 1990. User-Tailorable Systems: Pressing the Issues with Buttons. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*. ACM, 175–182. https://doi.org/10.1145/97243.97271

[45] John Maloney. 1995. Morphic: The Self User Interface Framework. *Self 4.0 Release Documentation* (1995).

[46] Andreas Mathisen, Tom Horak, Clemens N. Klokmose, Kaj Grønbæk, and Niklas Elmqvist. 2019. InsideInsights: Integrating Data-Driven Reporting in Collaborative Visual Analytics. *Computer Graphics Forum* 38, 3 (2019), 649–661. https://doi.org/10.1111/cgf.13717

[47] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-Rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proceedings of the ACM on Human-Computer Interaction* 3, 72, Article 72 (2019). https://doi.org/10.1145/3359174

[48] K. Jarrod Millman and Fernando Pérez. 2014. Developing Open-Source Scientific Practice. In *Implementing Reproducible Research*. CRC Press, 149–183. https://doi.org/10.1201/9781315373461-6

[49] Theodor Holm Nelson. 1995. The Heart of Connection: Hypermedia Unified by Transclusion. *Commun. ACM* 38, 8 (1995), 31–33. https://doi.org/10.1145/208344.208353

[50] Midas Nouwens, Marcel Borowski, Bjarke Fog, and Clemens Nylandsted Klokmose. 2020. Between Scripts and Applications: Computational Media for the Frontier of Nanoscience. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. https://doi.org/10.1145/3313831.3376287

[51] Midas Nouwens and Clemens Nylandsted Klokmose. 2018. The Application and Its Consequences for Non-Standard Knowledge Work. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. https://doi.org/10.1145/3173574.3173973

[52] Observable, Inc. [n.d.]. Observable. Retrieved December 31, 2021 from https://observablehq.com

[53] Fernando Pérez. 2013. "Literate computing" and computational reproducibility: IPython in the age of data-driven journalism. Retrieved December 31, 2021 from http://blog.fperez.org/2013/04/literate-computing-and-computational.html

[54] Project Jupyter. [n.d.]. Jupyter Notebook. Retrieved December 31, 2021 from https://jupyter.org

[55] Krishna Regmi, Jennie Naidoo, and Paul Pilkington. 2010. Understanding the Processes of Translation and Transliteration in Qualitative Research. *International Journal of Qualitative Methods* 9, 1 (2010). https://doi.org/10.1177/160940691000900103

[56] George G. Robertson, D. Austin Henderson, and Stuart K. Card. 1991. Buttons as First Class Objects on an X Desktop. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology (UIST '91)*. ACM, 35–44. https://doi.org/10.1145/120782.120786

[57] Mike Robinson. 1993. Design for unanticipated use.... In *Proceedings of the Third European Conference on Computer-Supported Cooperative Work (ECSCW '93)*. Springer, 187–202. https://dl.eusset.eu/handle/20.500.12015/2541

[58] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM. https://doi.org/10.1145/3173574.3173606

[59] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. 2017. Codestrates: Literate Computing with Webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM. https://doi.org/10.1145/3126594.3126642

[60] Christopher Schuster and Cormac Flanagan. 2015. Live Programming for Event-Based Languages. In *Proceedings of the 2015 Reactive and Event-based Languages and Systems Workshop (REBLS '15)*, Vol. 15. https://chris-schuster.net/rebls15/rebls2015-final.pdf

[61] Mark Stefik, Daniel G. Bobrow, Gregg Foster, Stan Lanning, and Deborah Tatar. 1987. WYSIWIS Revised: Early Experiences With Multiuser Interfaces. *ACM Transactions on Information Systems (TOIS)* 5, 2 (1987), 147–167. https://doi.org/10.1145/27636.28056

[62] Philip Tchernavskij, Clemens Nylandsted Klokmose, and Michel Beaudouin-Lafon. 2017. What Can Software Learn From Hypermedia?. In *Proceedings of the International Conference on the Art, Science, and Engineering of Programming (Programming '17)*. ACM. https://doi.org/10.1145/3079368.3079408

[63] Pierre Tchounikine. 2017. Designing for Appropriation: A Theoretical Account. *Human–Computer Interaction* 32, 4 (2017), 155–195. https://doi.org/10.1080/07370024.2016.1203263

[64] Mark Weiser. 1991. The Computer for the 21st Century. *Scientific American* (1991). https://doi.org/10.1038/scientificamerican0991-94

[65] Daisy Yoo, Peter Dalsgaard, Alix Ducros, Aurélien Tabard, Eva Eriksson, and Clemens Nylandsted Klokmose. 2020. Putting Down Roots: Exploring the Placeness of Virtual Collections in Public Libraries. In *Proceedings of the 2020 ACM Designing Interactive Systems Conference (DIS '20)*. ACM, 723–734. https://doi.org/10.1145/3357236.3395587

[66] Daisy Yoo, Aurélien Tabard, Alix Ducros, Peter Dalsgaard, Clemens Nylandsted Klokmose, Eva Eriksson, and Sofia Serholt. 2020. Computational Alternatives Vignettes for Place- and Activity-Centered Digital Services in Public Libraries. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. ACM. https://doi.org/10.1145/3313831.3376597