Codestrates: Literate Computing with Webstrates

Roman Rädle[1], **Midas Nouwens**[1], **Kristian Antonsen**[1], **James R. Eagan**[2,3,4], **Clemens N. Klokmose**[1]
[1]Aarhus University, [2]LTCI, [3]Télécom ParisTech, [4]Université Paris-Saclay
{roman.raedle,kba}@cc.au.dk, james.eagan@telecom-paristech.fr, {midasnouwens,clemens}@cavi.au.dk

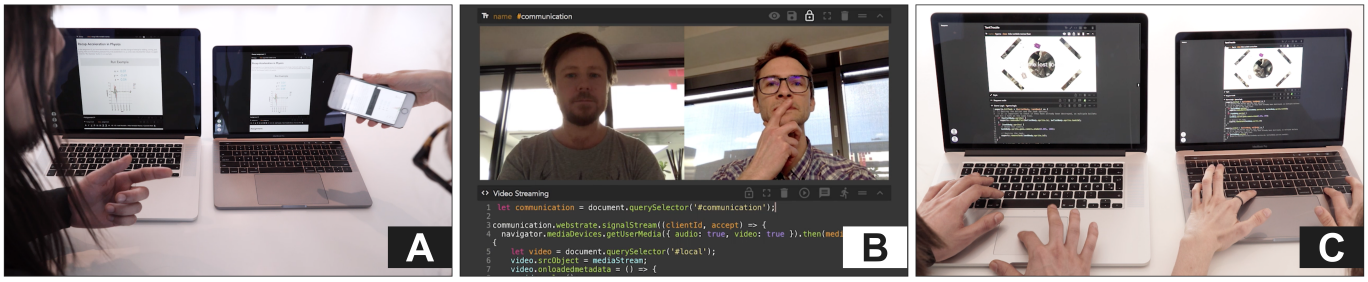

Figure 1. Three example uses of *Codestrates*. (A) Collaborative authoring of a physics report; accelerometer data from a phone is visualized in real-time in the codestrate and across multiple devices. (B) A codestrate is extended with real-time video communication. (C) Runtime tinkering with the mechanics of a game implemented in a codestrate.

ABSTRACT

We introduce *Codestrates*, a literate computing approach to developing interactive software. *Codestrates* blurs the distinction between the use and development of applications. It builds on the literate computing approach, commonly found in interactive notebooks such as Jupyter notebook. Literate computing weaves together prose and live computation in the same document. However, literate computing in interactive notebooks are limited to computation and it is challenging to extend their user interface, reprogram their functionality, or develop stand-alone applications. *Codestrates* builds literate computing capabilities on top of Webstrates and demonstrates how it can be used for (i) collaborative interactive notebooks, (ii) extending its functionality from within itself, and (iii) developing reprogrammable applications.

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

Author Keywords

Literate programming; literate computing; interactive notebooks; real-time collaboration; reprogrammable systems

INTRODUCTION

Webstrates [7] demonstrates how software can become reprogrammable and extensible in a collaborative fashion, blurring the distinction between applications and documents through a simple change to the web stack—making the Document Object Model (DOM) of web-pages persistent and collaboratively editable. Klokmose et al. present two approaches for developing with Webstrates: (i) using the web browser's built-in developer tools to edit the DOM, and (ii) using a dedicated code-editor webstrate that loads the code of other webstrates through *transclusion* using *iframes*. In both of these cases, there is a separation between using a webstrate and changing its appearance and behavior. The first approach limits development to desktop computers and makes use of a tool meant for debugging rather than developing. The second introduces an application-document relationship between webstrates, where the user has the overhead of loading the target webstrate in a separate code editor to make changes.

In interactive notebooks, use and development happen in the same context. They have become popular with non-professional programmers in education and scientific communities because they allow for authoring content, using code to process data, and visualizing results in the same document [8]. The creators of the popular Jupyter notebook call this approach *literate computing* [12]. It is based on *literate programming* [9] proposed by Knuth, which is a way to intermix code and textual narrative in a single document. Through the process of tangling and weaving, code and narrative are divided into separate files whereby code becomes executable, and the narrative presents itself in a human readable format. Knuth's goal was to "provide a tool for system programmers, not for high school students or for hobbyists" [9], where system programmers can explain programs "better than ever before."

Literate computing as realized in interactive notebooks combines prose and rich media with executable code. It allows everyone—including high school students and hobbyists— to weave "a narrative directly into a live computation, interleaving text with code and results to construct a complete piece that relies equally on the textual explanations and the computational components."[1] In an interactive notebook, documents

UIST 2017, October 22–25, 2017, Quebec City, QC, Canada

 ISBN 978-1-4503-4981-9/17/10…$15.00

DOI: https://doi.org/10.1145/3126594.3126642

[1]Definition by Fernando Perez — http://blog.fperez.org/2013/04/literate-computing-and-computational.html (last accessed: July 16, 2017)

can be created that interleave blocks of executable code with blocks of (rich) text. The output from the code can become itself part of the document (e.g., in the form of textual data or graphics).

The driving motivation for literate computing, as introduced with interactive notebooks like Jupyter, has been to improve scientific reproducibility and to integrate scientists' writing and computing activities in one environment [12]. We believe the literate computing approach of interactive notebooks has potential beyond scientific computing, especially for narrowing the gap between developing and using applications. However, today's interactive notebooks have limitations: (i) saving application states is difficult, limiting the ability to develop applications from within a notebook, (ii) real-time collaboration is, at best, limited to text editing, and (iii) the behavior of a notebook cannot be reprogrammed or extended from within, limiting its expressive power.

We present *Codestrates*, an alternative approach to building user-extensible collaborative interactive systems that combines literate computing with the possibilities of Webstrates. First, *Codestrates* pushes the literate computing approach by making possible the collaborative computation, extension, and development of applications with persistent states in the same environment, hereby narrowing the gap between development and use of interactive systems. Second, *Codestrates* enables prototyping in a manner similar to code playgrounds (e.g., Swift, Codepen, or JSFiddle), but with the potential to become usable applications by persisting their states. Third, *Codestrates* provides a development environment for Webstrates that goes beyond the paradigmatic application-document model. *Codestrates* is open source and ready for everyone to tinker with at `http://codestrates.org`.

We review related work, explain the concept of *Codestrates*, and demonstrate its capabilities with three usage scenarios, inspired by our own day-to-day use: (i) using a codestrate as a collaborative interactive notebook (Figure 1A), (ii) extending the functionality of a codestrate from within itself (Figure 1B), and (iii) developing reprogrammable applications in a codestrate (Figure 1C). We explain the technical implementation of *Codestrates*, discuss its limitations, and evaluate it based on Olsen's "solution viscosity" criteria [15].

RELATED WORK

Codestrates combines real-time, web-based collaborative authoring; documents that blend multimedia with executable code in a literate computing style; and end-user (re)programmability of document and application. We discuss related work that combines some of these elements.

Collaborative systems and documents

Jupiter [14] was an early collaborative multi-user dungeon built around shared, persistent "virtual places" (i.e., rooms). Users could create and customize places, documents, and tools in Jupiter using the provided high-level windowing toolkit or the internal programming language. Since then, Google Docs has established itself as one of the first web-based tools for real-time collaboration on documents. Dropbox recently released their own collaborative word processor called Paper[2], which goes beyond traditional documents by allowing users to write text, embed rich media, and include (non-executable) code snippets.

Scriptable and reprogrammable applications

Shareable and malleable applications

HyperCard [5] was an early hypermedia system for producing software that could easily be shared with and adapted by others. Through a visual drag-and-drop interface, end-users could create applications by building "stacks" of interactive cards. Users could programmatically add interactivity to the cards (e.g., a button on a card could link to another card in the stack) using the provided scripting language Hypertalk. However, HyperCard was only scriptable and not fully reprogrammable.

Reprogrammability of an environment at runtime

Smalltalk programming systems like Squeak [6] or Pharo[3] allow users to extend a program's functionality or even reprogram it entirely at runtime through just-in-time compilation and late binding [4]. Smalltalk relies on an image-based persistence model which forgoes a hard distinction between system code, application code, and application state. In the recent Lively project [18, 10], the concepts of Smalltalk have been ported to the modern web architecture using JavaScript. It takes an object-oriented approach to UIs based on Morphic [11] by abstracting over the DOM and CSS.

Web-based code playgrounds & reactive programming

Web-based programming environments like JSFiddle[4], JS Bin[5], and Codepen[6] allow the user to experiment with code and rapidly develop user interfaces, functionality, or applications that stay reprogrammable and can be shared with others. However, (i) the persistence and sharing of application states is not supported, (ii) collaboration is only possible for the editing of code, and (iii) the user's code cannot change the development environment (e.g., to increase its expressive match [15] by customizing its tools to match a programmer's personal preference).

Various web-based systems exist that try to make application development more approachable by going beyond traditional programming, such as through spreadsheet-like environments in Gneiss [2] or using only HTML in Mavo [19]. However, reprogramming the applications requires external editors or importing them back into the development environment, rather doing the edits within them.

Interactive notebooks using literate computing

Interactive notebooks

Jupyter notebook (formerly IPython Notebook)[7] and Apache Zeppelin[8] are two popular interactive notebooks that can embed code in multiple programming languages. Interactive

[2] `https://paper.dropbox.com/` (last accessed: July 17, 2017)
[3] `http://pharo.org/` (last accessed: July 17, 2017)
[4] `https://jsfiddle.net/` (last accessed: July 17, 2017)
[5] `https://jsbin.com/` (last accessed: July 17, 2017)
[6] `http://codepen.io/` (last accessed: July 17, 2017)
[7] `http://jupyter.org` (last accessed: July 17, 2017)
[8] `http://zeppelin.apache.org` (last accessed: July 17, 2017)

notebooks can be used for data cleaning and transformation, numerical simulation, statistical modeling, and machine learning. However, the scope of the code extends only to the content of the notebook; the user interface cannot readily be extended from within a notebook. Jupyter supports extensions, but these have to be installed on the server side and are developed externally to the notebook. Zeppelin supports collaborative editing, but only in designated text areas instead of the whole document. Neither Jupyter nor Zeppelin is designed for developing stateful applications; persisting data requires manually writing data to the file-system of the host computer or through a database interface.

Reprogrammable applications using literate computing

Leisure[9] and Eve[10] are systems that share the most with *Codestrates*. Leisure is an open source, web-based, and document-centric approach to computing based on the Emacs org-mode document format [17]. Leisure provides two-way bindings between an interactively editable representation of the document and its org-mode representation. Leisure documents are served statically from a web server (but can retain changes by connecting to a local Emacs buffer on a client's machine). Leisure supports WebSocket-based remote collaboration over a separate relay server.

Eve is an ambitious project that aims to reimagine programming for everyone. It introduces a new programming language in which the system state (including the UI) is addressable through queries. Eve takes a literate programming approach to developing full web applications and uses an interactive notebook style user interface. Currently, the Eve project is still in development and (although planned) has yet to implement real-time collaboration and distribution across devices.

Codestrates combines conceptual ideas and technical implementations of these related works; it adopts an image-based persistence model inspired by Smalltalk, where the image is the content of a webpage. Similar to Lively, *Codestrates* builds on modern web technology, but does not abstract away from the DOM and conventional web development. *Codestrates* follows the literate computing approach (and visual structure) of interactive notebooks and the block-like code representation of online programming playgrounds, but it goes beyond in-line computation to programming and reprogramming applications—including itself. *Codestrates* provides Google Docs style real-time collaboration, but (by leveraging the Webstrates platform) shares the entire webpage instead of just the editor buffer. Finally, *Codestrates* adopts the prototype based approach to re-purposing software made by others from HyperCard.

CODESTRATES OVERVIEW

A codestrate is essentially a webpage whose content, presentation, and behavior can be (collaboratively) edited from within the page and whose edits are inherently made persistent. It includes everything it needs to both implement and edit itself. New codestrates are created by copying another codestrate and are automatically versioned. A codestrate can be broken into three components: *paragraphs*, *sections* of related paragraphs, and the entire codestrate implementation (details are in the implementation section).

[9] https://github.com/zot/Leisure (last accessed: July 17, 2017)

[10] http://witheve.com (last accessed: July 17, 2017)

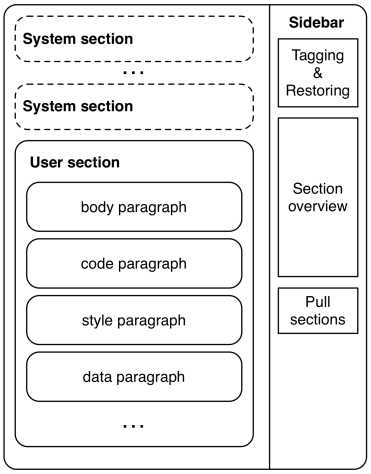

Figure 2. A schematic overview of the structure of a codestrate. On the left are sections, which include system sections (hidden by default) and one or more user sections. Sections can include paragraphs of different types (body, code, style, data). On the right is a sidebar (hidden by default), which contains actions for the codestrate (create a copy of the codestrate, tag and restore, pull from another codestrate) and actions for the sections (toggle sections' visibility, add section).

Use of paragraphs and sections

Paragraphs and *sections* in a codestrate are structured in a linear fashion, similar to traditional text documents. Figure 2 shows a schematic overview of the structure of a codestrate.

Paragraph types and their function

A *paragraph* can be of the type body, code, style, or data:

- Body paragraphs contain what is typically considered the content of a webpage and are directly editable through a simple rich-text editing interface or an HTML inspector (as illustrated in Figure 3).
- Code paragraphs contain editable JavaScript code that can be toggled to run on page load or be executed by pressing an execute button. The code in *Codestrates* executes in the runtime of the browser. Code paragraphs have syntax highlighting, indentation, auto-completion, and an expandable interactive console for debugging.
- Style paragraphs contain Cascading Style Sheet (CSS) rules. Changes are immediately reflected on the page. They have syntax highlighting, indentation, and auto-completion.
- Data paragraphs contain editable data in the JavaScript Object Notation (JSON) format and have syntax highlighting and indentation.

Each paragraph can be expanded to full-screen, collapsed to a header only, locked against edits, deleted, and moved up or down in the list of paragraphs or across sections. They can also have a title and are addressable in JavaScript or in CSS through their (optional) IDs or classes.

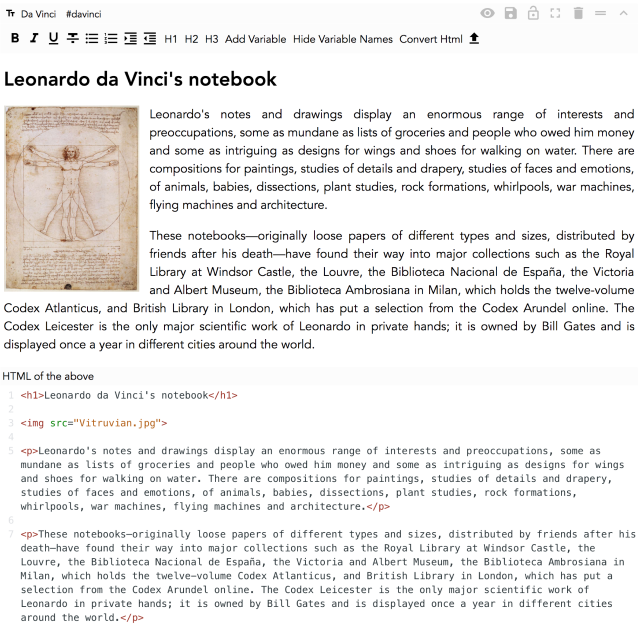

Figure 3. A screenshot of a codestrate; the user edited its style to favor a lighter appearance. It shows a body paragraph with its HTML inspector visible—visibility is toggled through the eye icon in the paragraph's header.

In *Codestrates*, the result of evaluating a code paragraph does not have a standard output. Instead, the idea is to output evaluated code results into body paragraphs. To facilitate this, a body paragraph can include variables whose value can be set from code paragraphs through a simple API (more details are in the implementation section).

Combining paragraphs in sections

Sections are collections of related paragraphs. We distinguish between *system* sections and *user* sections. System sections contain the implementation of the codestrate itself and are hidden by default. User sections contain whatever the user is working on. However, whether a section belongs to the user or the system is not fixed. Extending a codestrate with new functionality is essentially turning a user section into a system section by ticking a checkbox in the section's header.

All sections are listed in the sidebar, which also provides functions such as creating a new section, toggling a section's visibility, pulling sections from another codestrate, and tagging and restoring the codestrate.

The traditional boundaries between development and use are blurred in *Codestrates*. The user can fluidly move between the two to the extent that developing and using interactive systems are no longer necessarily separate activities. Figure 4 illustrates how a grocery list app can be developed in a codestrate with the user interface expressed in a body paragraph and a style paragraph, and the interaction implemented in a code paragraph that is set to run on page load (green running man). Setting the body paragraph to full-screen turns the codestrate into a "regular" application, usable across devices (since the state is synchronized through the DOM). If changes to the application are required, users can exit the full-screen

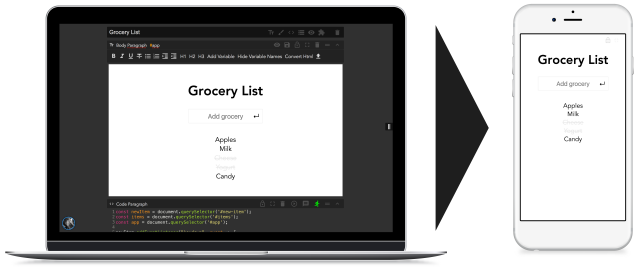

Figure 4. A simple grocery list implemented in a codestrate. On the left, the codestrate in a desktop browser showing a body paragraph and the top of a code paragraph. To the right, the same codestrate opened on a smartphone, the body paragraph has been made full-screen and is now functioning as a grocery list app.

mode again and make edits to the code—collaboratively and in real-time.

USES OF CODESTRATES

We present three scenarios that highlight *Codestrates*' capabilities, inspired by our own daily use over the course of six months. The scenarios assume a future in which teachers and students are fluent in computational thinking [20] and can master a medium that requires more technical knowledge than what is common today. The scenarios are also demonstrated in the supplementary video.

Interactive notebooks in Codestrates

Alex, a secondary school physics teacher, prepares an assignment on speed and acceleration. He provides the students with a notebook for the assignment, including interactive code examples. One of the examples shows how students can access the accelerometer and GPS data from their mobile phones (Figure 1A). Later, on their laptops, the students add a button that uses Alex's code sample to store their current position and acceleration in a data paragraph. They then collect data by opening the codestrate on a smartphone and walking, running, and cycling outdoors. When they return, they plot a graph of the captured data for another assignment in the notebook.

How it works

The assignment notebook is created by copying a codestrate (e.g., by opening the URL `/codestrate/?copy=assignment-notebook`), adding a section and a body paragraph to it, and using the rich-text capabilities to add content. Interfacing with the sensors on the device is done by writing JavaScript in a code paragraph and using the geolocation[11] and devicemotion[12] APIs. The sensors' current values are displayed through variables, which are inserted into a body paragraph and set through JavaScript in a code paragraph. The newest version of the codestrate is then tagged as stable and shared as a URL (e.g., `/assignment-notebook/stable/`, where stable is the tag name).

Users create their copy of the stable version using the same copy mechanism as before, but with the tagged assignment

[11] `https://developer.mozilla.org/en-US/docs/Web/API/Geolocation` (last accessed: July 17, 2017)

[12] `https://developer.mozilla.org/en-US/docs/Web/Events/devicemotion` (last accessed: July 17, 2017)

notebook as source codestrate (e.g., `/assignment-notebook/stable/?copy`). A data paragraph is added to the copy and the existing code paragraph is edited to store sensor values in the data paragraph. The HTML editor of a body paragraph can be used to add a button and an additional code paragraph to make the button interactive. To plot data, the codestrate functionality needs to be extended either by writing code for custom visualization or by importing data visualization libraries (e.g. Vega Lite[13] [16]). The implementation section contains details on how to import libraries.

In real use

We have actively used *Codestrates* as an interactive assignment environment in our own classes. Twenty-five students in an introduction to programming course for non-computer science students completed their hand-ins for five consecutive weeks using *Codestrates*. We prepared assignments as codestrates with written instructions, interactive examples, automated testing, and code scaffolds (i.e., working but incomplete code the students needed to edit). The assignment codestrates had a low threshold [13] and allowed our students to immediately start programming because they did not have to install development environments or fiddle with server settings and file uploads. We received positive feedback from the students through a dedicated feedback section in the codestrates they were using.

Extending codestrates in Codestrates

Alex notices that his students are collaborating remotely on their assignments after class. To hone his programming skills, he decides to extend the assignment codestrate with video communication. He copies a new codestrate and starts tinkering with *Web Real-Time Communications* (WebRTC). After a few iterations, Alex manages to stream audio and video from his web camera to all clients who have the same codestrate open (illustrated in Figure 1B). The codestrate already shows an avatar for each connected client in the bottom left corner, so Alex overlays users' avatars to display their video streams. He pulls the section with the WebRTC code into the assignments codestrate and emails the students to let them know they can update their codestrates if they want to use this new functionality.

How it works

For video and audio streams, users can exploit the getUserMedia[14] API. The video and audio stream can be added to a DOM node with Webstrates' streaming API[15]. To build the UI of this new functionality, the user can add HTML, CSS, and JavaScript in body, style, and code paragraphs respectively. The video element can be placed on top of the already existing avatar elements. The user can tag the current version of the codestrate with a meaningful name through an action in the sidebar. If an existing tag is reused, it will be updated to the current version. Other codestrates can update their sections with the changes using the "pull sections" function.

In real use

We have implemented several features by copying codestrates and pulling sections with new features back into the master codestrate. We usually started features in user sections but eventually changed them to system sections (e.g., remote pointers, video communication). The fluid way with which we can move between using and developing a codestrate results in the continuous development of functionality in response to specific tasks at hand. If the same functionality in one codestrate is needed at a later point in time for different tasks, those sections can easily be transferred between codestrates. For example, one user built a presentation tool for a research talk. That tool was then copied by someone else and extended with an in-slide code editor to teach a programming course. A static PDF viewer was extended to a mobile note-taking tool that allowed hand-written annotations from an iPad, then extended again into a review writing tool with a simple text processor next to it. Through using and developing codestrates this way, we have access to an organically growing repository of functionality instead of a limited collection defined during the traditional development phase.

Developing applications in Codestrates

Jim and Bethany, two of Alex's students, are part of an extracurricular game development club. They have been working on a multi-player tank game in a codestrate with the help of an open source web-based game engine. One day, while playing the game across their networked computers, they realize something is wrong with the physics of the bullets—not bouncing off the walls. They exit the full-screen mode of the body paragraph that hosts the game view and together edit the function that calculates the bullet path. The changes are immediately reflected in the running game, helping them iterate through different equations until they are satisfied (illustrated in Figure 1C).

How it works

Web-based game engines such as Phaser[16] can be used to render the game to a canvas element in a body paragraph. The game itself can be built using code paragraphs and is similar to regular game development in JavaScript. To add a multi-player mode in which the game state is synchronized between multiple players, users can—similar to the video streaming from above—leverage Webstrates' signalling API to send messages between clients of the same codestrate. It is possible to require the code of another code paragraph in the game loop, which allows for changing the game mechanics at runtime without reloading the page. Requiring code is explained in the implementation section.

In real use

We hired a professional game developer for a day to demonstrate that it was possible to build a multi-player game in a codestrate. He implemented a simplified version of a tank game he had previously developed[17]. With only a brief introduction to how *Codestrates* works and the signalling API of Webstrates, he was able to implement the game without significant assistance over the course of a day. He struggled with

[13] `https://vega.github.io` (last accessed: July 17, 2017)

[14] `https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia` (last accessed: July 17, 2017)

[15] `https://github.com/Webstrates/Webstrates` (last accessed: July 17, 2017)

[16] `https://phaser.io` (last accessed: July 17, 2017)

[17] `https://www.tanktrouble.com/` (last accessed July 11, 2017)

the lack of screen real estate to get an overview of all his code, but enjoyed the ability to tweak gameplay mechanics and see them reflected in the game immediately. We will address the lack of screen real estate in the discussion section.

IMPLEMENTATION

Codestrates is implemented on top of Webstrates [7]. In order to understand the architectural choices and implementation details of *Codestrates*, it is necessary to understand how Webstrates works.

How Webstrates works

Webstrates consists of a web server where all client-side changes to the DOM are persistent, and it synchronizes those changes to all clients of the same page. When a browser requests a page from the Webstrates server (e.g. **`/myWebstrate`**), it is served a generic HTML page containing a Webstrates client written in JavaScript. The client connects to the server through a web socket and receives the requested webstrate (e.g. myWebstrate) in a serialized JSON format. The client deserializes the JSON, populates the DOM of the generic HTML page, observes the DOM for changes using a `MutationObserver`, and listens on the web socket connection for changes made by other clients. Synchronization happens through operational transformation (OT) [3] using the open source OT framework ShareDB[18]. Webstrates uses OT to maintain a consistent document state across clients, thus providing real-time collaborative editing of the DOM. Because ShareDB synchronizes operations on JSON documents, Webstrates' inner representation of the DOM is JSON using JsonML[19]. Webstrates leverages the principle of transclusion (using *iframes*) as a composition mechanism, which creates a dynamic relationship between two or more webstrates.

A new webstrate is created by requesting a webstrate that does not exist or by creating a copy of an existing webstrate. A webstrate can be copied either to a new webstrate with a random id (using **`/myWebstrate/?copy`**) or to a named webstrate (using **`/myWebstrate/?copy=myWebstrateCopy`**).

Webstrates has a simple versioning mechanism based on operation logs. For example, requesting **`/myWebstrate/1432`** will retrieve the HTML of the 1432nd version of myWebstrate, and **`/myWebstrate/?restore=1432`** will restore it to the state it had at the 1432nd version by applying the operations matching the difference between the current and the 1432nd version. Versions can be tagged with human-readable names and tags can be retrieved in the same manner as versions (e.g., **`/myWebstrate/myTag`**). The Webstrates server automatically generates a tag for a webstrate when no edits were made for a set period of time.

Webstrates uses external authentication providers (e.g., GitHub). User rights such as read, read-write, or none can be added to a webstrate using the `data-auth` attribute on the `html` element. User information such as their username and avatar are accessible through a JavaScript API.

[18] **`https://github.com/share/sharedb`** (last accessed: July 17, 2017)

[19] **`http://www.jsonml.org`** (last accessed: July 17, 2017)

```
<html>
 <head>
  <script type="text/javascript">
   <!-- see Listing 2 -->
  </script>
 </head>
 <body>
  <div class="section section-hidden" data-type="system">
    <div id="bootstrap" class="paragraph code-paragraph">
     <pre type="text/javascript">
      // Code that queries and executes all
      // run-on-load code paragraphs
     </pre>
    </div>
  </div>
  <div class="section" name="A System Section"
       data-type="system">
    <div class="paragraph code-paragraph"
         run-on-load="true">
     <pre id="code-on-load" type="text/javascript">
      // Executed by code paragraph in line 10
     </pre>
    </div>
  </div>
  <div class="section" name="A User Section">
   <div class="paragraph body-paragraph">
    <div id="body" class="class1 class2"
         contenteditable="true">
     <!-- HTML -->
    </div>
   </div>
   <div class="paragraph style-paragraph">
    <style id="style" type="text/css">
     /* CSS */
    </style>
   </div>
   <div class="paragraph code-paragraph">
    <pre id="code" type="text/javascript">
     // JavaScript
    </pre>
   </div>
   <div class="paragraph data-paragraph">
    <pre id="data" type="application/json">
     /* JSON */
    </pre>
   </div>
  </div>
 </body>
</html>
```

Listing 1. Simplified HTML structure of a codestrate.

Extensions to Webstrates

In parallel to *Codestrates*, Webstrates has been extended with a number of features that make the development of larger systems more convenient. *Codestrates* relies on many of these extensions, e.g., tagging, transient elements, assets, signalling, and streaming (e.g., to support WebRTC).

A `transient` element has been introduced that allows for adding elements to the DOM that are neither persistent nor synchronized to other clients. A custom context menu is an example of something that makes sense to put in a transient element because it is ephemeral and only relevant for the user who opened it. In addition, users can exploit Webstrates' `transient` element in a codestrate to write an application that has visually distinct appearances on different clients.

Binary assets such as images or videos can be attached to a webstrate (through a POST request) and accessed as child web documents, e.g., **`myWebstrate/myVideo.mp4`**. Assets are versioned as with any other change made to the webstrate document.

A signalling API has been developed to allow clients of the same webstrate to communicate with each other without manipulating the DOM. Clients can broadcast signals to every client of the same webstrate or send targeted signals to a subset of clients (the client list is accessible through the API). Signals can contain JSON objects as message payloads. For example, *Codestrates* uses signals to communicate ephemeral states such as remote cursor and pointer positions or to establish WebRTC connections between clients of the same codestrate. Finally, streaming allows peer-to-peer communication between clients using WebRTC.

Codestrates

Every codestrate is a webstrate that includes the codestrate implementation and the user content: it is completely self-contained. The markup (HTML), styling (CSS), program code (JavaScript), and data (JSON) are stored in a codestrate's document body, each wrapped in a paragraph element (`<div class="paragraph">`). As a structuring mechanism, paragraphs are grouped inside sections (`<div class="section">`). Webstrates ensures that changes to the content of a codestrate are made persistent and synchronized to all clients of the same codestrate. Listing 1 shows a simplified HTML of the document structure with two system sections (one hidden), a user section, three code paragraphs (one hidden by its section), a body paragraph, a style paragraph, and a data paragraph.

Bootstrapping and code execution

Code execution in *Codestrates* differs from the regular JavaScript execution routine of a browser. Its execution relies on three integral conditions: (i) there is bootstrap code in a `script` element in the document `head` (Listing 1, line 3); (ii) all other code is stored in `pre` elements, which are not executed on page load; and (iii) one code paragraph has the id `#bootstrap` (Listing 1, line 9).

A codestrate is bootstrapped with the four lines of code in Listing 2, which are executed after the webstrate has loaded. This code triggers the execution of the code paragraph with the id `#bootstrap`, which queries the DOM for all other code paragraphs with the attribute `run-on-load` set to `true` (e.g., Listing 1, line 19) and synchronously executes them in the order that they appear in the DOM.

```
1 webstrate.on("loaded", function () {
2  var codeParagraph = document.querySelector("#bootstrap");
3  new Function(codeParagraph.textContent)();
4 });
```

Listing 2. *Codestrates*' bootstrap JavaScript code.

Each code paragraph is executed in its own scope and execution context using the `Function` object. Code paragraphs do not create closures to their creation contexts and are only able to access their own local variables and variables defined in global scope (e.g., the `document` object to call the `querySelector` function). This prevents users from accidentally overriding and interfering with *Codestrates*' execution logic. The execution context provides access to a proxied console and a `Variable` object. Each code paragraph has its own console output; all `log`, `error`, `debug`, `warn`, and `info` function calls on the console object are logged to this output before they are redirected to the window's console. The `Variable` object provides convenience functions to replace content in a body paragraph. For example, a variable *myVar* is inserted into a body paragraph using its rich-text editor tools (see *Add Variable* in Figure 5). The content of *myVar* can be set from any code paragraph using `Variable("myVar").set("newValue")`. The variable is represented as `<div class="variable" data-name="myVar"></div>` in the body paragraph.

Requiring modules and importing external libraries

A code paragraph can require the execution of another code paragraph using CSS selectors (e.g., `var myModule = require("#myModule")`) or class selectors (e.g. `require(".myModules")`). When code is required using a class selector, all code paragraphs of that particular class are queried, their contents are concatenated, and the code is executed as one script. Executing the content of a required code paragraph adds an additional `exports` object in the execution context. A *required* code paragraph can expose variables and functions to the caller paragraph using the exports object (e.g., `exports.myVar = "myValue"` or `exports.myFunction = function() {...}`). The exports object is a plain key value store that allows code paragraphs to export multiple variables and functions. The require function will return the exports object in order for the calling code paragraph to access exported variables and functions (e.g., `myModule.myVar` or `myModule.myFunction()`).

External JavaScript libraries can be used through `importLib` (Listing 3), which takes either a single URL or reference to a webstrate asset, or an array of them. For each URL, *Codestrates* adds a `transient` element to the document `head` with the `script` element inside. This way, added scripts do not persist and are not synchronized with other clients. Because browsers execute `script` elements asynchronously when added at runtime, two external JavaScript libraries that depend on each other can cause faulty code when added after page load. *Codestrates*' `importLib` guarantees that external JavaScript libraries are loaded and executed synchronously in the order in which they have been defined in the array.

The `importLib` function returns a promise which resolves after all libraries have been imported (i.e., loaded and executed). The code in the resolve function can then use imports the same way as `script` elements that are loaded synchronously. Importantly, subsequent code paragraphs will be blocked until the preceding code paragraph has *imported* and executed all external libraries, executed all code paragraphs that are *required* within, and finally executed all of its own code.

```
1 importLib([
2   "//cdn/extLibrary.js",
3   "assetLibraryDependingOnExtLibrary.js"
4 ]).then(() => {
5   // code executes after both libraries are imported
6 });
```

Listing 3. Import external libraries in a *Codestrates* code paragraph.

Generating user interfaces and structuring paragraphs' data

As part of *Codestrates*' core system functionality, the user interface (UI) elements for sections and paragraphs are generated at runtime (e.g., the rich-text editor tools for a body paragraph

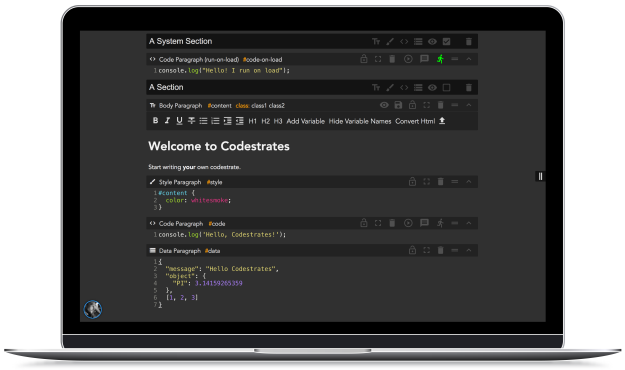

Figure 5. Codestrate view of Listing 1, including paragraphs and their contents. The section containing the bootstrap code is hidden.

(Figure 5)). A `MutationObserver` observes the `body` element of a codestrate and its subtree to create user interfaces for sections and paragraphs that are added to the document after initial loading. The UI elements are programatically created using `transient` elements in order to keep their states local instead of synchronizing them between clients. This way, expanding the HTML editor of a body paragraph only has an effect locally.

The content of body paragraphs are `contenteditable`[20] `div` elements directly visible to the user. All changes within a `div` element are immediately reflected in the DOM and Webstrates synchronizes changes with other users of the same codestrate.

For the code, style, and data paragraphs, the element storing their content is hidden from the users through CSS, and a `transient` element is generated with a CodeMirror[21]-based editor that creates a two-way binding between its text buffer and (remote) changes to the element's content. CodeMirror comes with support for syntax highlighting, code completion, formatting, and other functionality expected from modern code editors. As an optimization, we only instantiate CodeMirror instances for visible paragraphs.

CSS is stored directly in a `style` element. Therefore, changes to CSS rules in style paragraphs have an immediate effect on the rendering of the codestrate's content. Code is executed either explicitly by the user through pressing the execute button in the code paragraph's header, or after the webstrate has been loaded provided that the user enabled the run-on-load in the code paragraphs header (the green running man Figure 5, top).

Remote collaboration

Since Webstrates synchronizes the DOM between connected clients transparently, *Codestrates* allows for remote collaboration. However, ephemeral data such as pointers (e.g., mouse cursor or touch points) are not synchronized.

Using signalling, we implemented a user manager and remote pointers in *Codestrates* to support awareness of other users. Every client that joins a codestrate broadcasts the user's information to all other clients. If a user is authenticated, the codestrate broadcasts the username, friendly name, and avatar url of the user; otherwise it broadcasts as anonymous with a default avatar. The default codestrate UI shows all connected users as avatars in the bottom left corner in a `transient` element. Each user gets assigned a distinct color that is visible as the border of their avatar.

Each client broadcasts pointer positions (such as `mousemove` and `touchmove`) and pointer actions (such as `click` and `tap`). Remote pointers are represented as `transient` elements positioned relative to the nearest paragraph, which share dimensions across devices and screen sizes. Actions are styled `div` elements appended to the pointer element and removed after a timeout. Cursor positions and selections in editors are synchronized across clients to allow for Google Docs-like document-centric authoring. The user color from the user manager is used to color pointers and cursors in order to associate actions with remote users.

Video and audio communication is implemented using Webstrates' `transient` element and streams, and the `getUserMedia` API. A codestrate listens for incoming streams from other clients, automatically accepts them, and overlays the sender's avatar with a `video` element rendering the stream. The `video` element is wrapped in a `transient` element and not synchronized with other clients. To start video and audio streaming, a user has to click on the video camera icon that is revealed when hovering over the user's own avatar.

Versioning and updating

Codestrates provides a UI for easily tagging and restoring versions of a codestrate. Pressing the restore button in the sidebar shows a dialog with a list of both user-generated and auto-generated tags, which are accessed using the Webstrates versioning API.

Because a codestrate contains all of its implementation, it needs to be updated manually when a new feature is introduced or a bug is found in the codestrate from which it was copied. To support this, users can pull sections from other codestrates or even from an earlier version of the same codestrate. The pull sections functionality loads a codestrate of a specific version into the calling codestrate using an `iframe` wrapped in a `transient` element. To pull a section, the user provides the codestrate id and (optionally) specifies the version of the codestrate to be pulled and a CSS selector for particular elements of the codestrate (if unspecified, the latest version and the "system section" selector will be used). The codestrate then deletes all of its sections that match the CSS selector; queries sections matching the CSS selector in the codestrate being pulled and deep clones them; appends the cloned elements to itself; and finally reloads the webpage after all operations (OT) are synchronized with the server. While pulling sections would work without a reload, doing so ensures code written by others always runs without side-effects.

[20] **https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/contenteditable** (last accessed: July 17, 2017)

[21] **http://codemirror.net** (last accessed: July 17, 2017)

DISCUSSION

Codestrates is a proof of concept that targets the complex activity of both using and developing interactive systems. Some limitations arise due to the wide variety of practices that modern development environments are expected to support (e.g., fine-grained version control and interoperability with the operating system). Thus, a comparative evaluation with well-established systems (e.g., development environments or code playgrounds) would be challenging, as stated by Olsen [15]. He proposes "solution viscosity" criteria as alternatives, which allow user interface systems research to be evaluated analytically. We discuss *Codestrates* as a literate computing approach to developing interactive systems using these principles.

Limitations and future work

Out-of-browser code execution and Jupyter integration

Code execution in *Codestrates* happens at runtime and only within the browser. This limits language support to JavaScript (or languages compiled or transpiled to it) and means it does not have access to the operating system of the host computer. However, we have successfully experimented with using a Jupyter kernel running on the local machine to execute Python code from a codestrate. We created Python specific code paragraphs that post their code over HTTP to the local Jupyter kernel when executed and have the standard output redirected to the console of the code paragraph. Future research includes a model for managing in-browser and out-of-browser computation (e.g., on a local computer or an online service) and polyglot code execution in a codestrate where various programming languages can be combined to perform a series of dependent computations.

Version control

Version control systems (e.g., Git or Subversion) are essential for systems development. *Codestrates* allows tagging of and branching from codestrates, but updating from another codestrate replaces affected paragraphs entirely. It does not support automatic or manual merging of paragraph contents. We plan to integrate merging algorithms in future work, such as a recursive three-way merge.

Codestrates' "pull section" mechanism allows for very idiosyncratic ways of feature and application development: (i) changes can happen in a copy of the *Codestrates* prototype and after testing be pulled back into the prototype; (ii) copies can be updated to the latest version of the *Codestrates* prototype by pulling the latest version of system sections; and (iii) (system) sections in a codestrate can be downgraded without *undoing* the changes in-between (e.g., the changes that happened to other sections).

Overhead of development environment

Applications developed with *Codestrates* carry more weight than regular web applications because they include both the application code and a development environment. Loading time and memory consumption of an application built with *Codestrates* is higher than a comparable traditional web application. The overhead adds ~800 kilobytes of data (including external and non-minified libraries like CodeMirror). Future work includes caching strategies for external libraries and lazy loading of the development environment.

Textual programming

Codestrates currently only allows users to express interactive behavior through textual programming using JavaScript and the DOM API. Visual languages, such as Scratch[22], or languages influenced by natural language, such as HyperTalk [5], have successfully introduced beginners and children to programming. These approaches could be integrated in *Codestrates* and even be used interchangeably, so that *Codestrates* can adapt to the experience level of the programmer. We strongly believe that the threshold for expressing interaction and computation in a codestrate should be lower so it can be used by a wider audience with different levels of computational literacy [13].

Usability

Full-screening a paragraph is currently not a transient action, which means that it affects all clients of the same codestrate. This makes it impossible for a user to change the style or code of a codestrate while also running it as a full-screen application on another device or for multiple users to view the codestrate in different ways. Adding this flexibility would mean that users could maximize the use of screen real-estate (e.g., across multiple screens instead of continuously scrolling through the document), which was mentioned as a limitation by the game developer who implemented the multi-player game in *Codestrates*. This would be possible by CSS rules targeting transient attributes available with the newest version of Webstrates, attributes that are not persistent and synchronized.

While the CodeMirror library we use in *Codestrates* provides many of the editing capabilities of modern programming IDEs, it is still limited compared to desktop editors. For example, it lacks advanced auto-completion and refactoring features now taken for granted when developing software. (Their absence was noticed by our students.)

Systems-oriented evaluation

According to Olsen, a user interface system can be evaluated according to its "solution viscosity", which expresses the effort of a programmer to create possible solutions. Good systems can reduce solution viscosity in three ways: *flexibility*, *expressive leverage*, and *expressive match*.

Flexibility

Good flexibility allows the user to make rapid changes and evaluate them immediately [15]. *Codestrates* supports flexibility with paragraphs and sections that remain inspectable and whose changes can be evaluated at runtime. For example, a user can customize the look of a codestrate-based application by changing CSS properties in a style paragraph, change application behavior by changing the respective code paragraph, or add functionality at runtime by writing new code.

Expressive leverage

A system with a high expressive leverage reduces the choices a user can make while still being able to express more [15]. Since *Codestrates* builds on Webstrates, all content is by default persistent and synchronized. As a result, there are no

[22] https://scratch.mit.edu/ (last accessed: July 17, 2017)

additional software layers necessary to add real-time collaboration and there is no need to create databases or add services to persist application data and states.

Expressive match

The expressive match refers to "how close the means for expressing design choices are to the problem being solved" [15]. With *Codestrates*, we anticipate a future generation where computational thinking is part of formal education practices and considered a "fundamental skill for everyone" [20]. Although this generation will know how to express computation in code, they are considered non-professional programmers [1] and may not have acquired a fundamental knowledge about computing technology (e.g., an understanding of client/server communication). Olsen argues that new tools should be "accessible, easier or more effective for this desired population" and they should support "different norms of expression or design goals that are not supported by existing tools."

With literate computing expressed in *Codestrates*' paragraphs, users can "read a program" from top to bottom to understand its execution order, much like reading a book or an article. This is greatly different from often complex dependencies in file-based projects. It is also different from a single HTML file containing HTML, JavaScript, and CSS edited in a file editor; paragraphs in *Codestrates* containing HTML, JavaScript, or CSS are visually distinct from each other and provide additional tools like "execute" for code paragraphs or rich-text editor tools for body paragraphs. Changes to body, style, code, or data paragraphs are immediately part of the codestrate that contains them and thus are already "deployed." There is no need to copy code from a playground, paste it to a file, and deploy it on a server to make it accessible to everyone. *Codestrates* is agnostic to any programming pattern and might in the future even be agnostic to programming language (we are currently investigating this topic). Users can exploit their pre-existing knowledge on web development without the need to learn or adapt to a new programming language.

CONCLUSION

We have demonstrated how literate computing with Webstrates not only allows for mixing prose and computation, but also for extending the functionality of interactive notebooks from within itself and developing reprogrammable applications—collaboratively.

Codestrates is built on Web standards and is easily appropriable and usable by anyone with a basic Web development background. However, the presented usage scenarios in this paper are based on currently fictional levels of technical proficiency of non-professional programmers. Future research includes iteratively co-designing codestrates to support real users such as data scientists, secondary school teachers, students, and hobbyists, and exploring what could aid or prevent end-user adoption of a medium such as *Codestrates*.

ACKNOWLEDGEMENTS

This work has been funded by the Aarhus University Research Foundation. We thank Wendy Mackay, Michel Beaudouin-Lafon, and the anonymous reviewers for their valuable suggestions and comments, Romain Primet for experimental Jupyter integration, Brian Bunch Christensen for implementation of the tank game, Jonas Oxenbøll Petersen for AV assistance, and Lindsay Reynolds for proof-reading this paper.

REFERENCES

1. Margaret M. Burnett and Brad A. Myers. 2014. Future of End-user Software Engineering: Beyond the Silos. In *Proceedings of the on Future of Software Engineering (FOSE 2014)*. ACM, New York, NY, USA, 201–211. DOI:http://dx.doi.org/10.1145/2593882.2593896
2. Kerry Shih-Ping Chang and Brad A. Myers. 2014. Creating Interactive Web Data Applications with Spreadsheets. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 87–96. DOI: http://dx.doi.org/10.1145/2642918.2647371
3. C. A. Ellis and S. J. Gibbs. 1989. Concurrency Control in Groupware Systems. *SIGMOD Rec.* 18, 2 (June 1989), 399–407. DOI: http://dx.doi.org/10.1145/66926.66963
4. Adele Goldberg. 1995. Why Smalltalk? *Commun. ACM* 38, 10 (Oct. 1995), 105–107. DOI: http://dx.doi.org/10.1145/226239.226260
5. Danny Goodman. 1987. *The Complete HyperCard Handbook*. Random House Inc., New York, NY, USA.
6. 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. *SIGPLAN Not.* 32, 10 (Oct. 1997), 318–326. DOI: http://dx.doi.org/10.1145/263700.263754
7. 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, New York, NY, USA, 280–290. DOI: http://dx.doi.org/10.1145/2807442.2807446
8. Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, and et al. 2016. Jupyter Notebooks–a publishing format for reproducible computational workflows. *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (2016), 87–90. DOI: http://dx.doi.org/10.3233/978-1-61499-649-1-87
9. Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111. DOI: http://dx.doi.org/10.1093/comjnl/27.2.97
10. Robert Krahn, Dan Ingalls, Robert Hirschfeld, Jens Lincke, and Krzysztof Palacz. 2009. Lively Wiki a Development Environment for Creating and Sharing Active Web Content. In *Proceedings of the 5th International Symposium on Wikis and Open Collaboration (WikiSym '09)*. ACM, New York, NY,

USA, Article 9, 10 pages. DOI: http://dx.doi.org/10.1145/1641309.1641324

11. John H. Maloney and Randall B. Smith. 1995. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology (UIST '95)*. ACM, New York, NY, USA, 21–28. DOI: http://dx.doi.org/10.1145/215585.215636
12. Jarrod K. Millman and Fernando Pérez. 2014. Developing open-source scientific practice. *Implementing Reproducible Research* 149 (2014).
13. Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (March 2000), 3–28. DOI: http://dx.doi.org/10.1145/344949.344959
14. David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. 1995. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology (UIST '95)*. ACM, New York, NY, USA, 111–120. DOI: http://dx.doi.org/10.1145/215585.215706
15. Dan R. Olsen, Jr. 2007. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, New York, NY, USA, 251–258. DOI: http://dx.doi.org/10.1145/1294211.1294256
16. Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (jan 2017), 341–350. DOI: http://dx.doi.org/10.1109/tvcg.2016.2599030
17. Eric Schulte and Dan Davison. 2011. Active Documents with Org-Mode. *Computing in Science & Engineering* 13, 3 (may 2011), 66–73. DOI: http://dx.doi.org/10.1109/mcse.2011.41
18. Antero Taivalsaari, Tommi Mikkonen, Dan Ingalls, and Krzysztof Palacz. 2008. *Web Browser As an Application Platform: The Lively Kernel Experience*. Technical Report. Mountain View, CA, USA.
19. Lea Verou, Amy X. Zhang, and David R. Karger. 2016. Mavo: Creating Interactive Data-Driven Web Applications by Authoring HTML. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 483–496. DOI: http://dx.doi.org/10.1145/2984511.2984551
20. Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (March 2006), 33–35. DOI: http://dx.doi.org/10.1145/1118178.1118215