Rethinking Interaction with Literate Computing

Clemens N. Klokmose
Aarhus University
Aarhus, Denmark
clemens@cavi.au.dk

Roman Rädle
Aarhus University
Aarhus Denmark
roman.raedle@cc.au.dk

Abstract
Given the wide uptake of interactive notebooks such as Jupyter Notebook, the software paradigm of literate computing have in recent years gained significant popularity. In this position statement, we will unfold our experiences of working with literate computing. We draw these experiences from a one-and-a-half year of using Codestrates—our implementation of a literate computing environment.

Author Keywords
Interactive notebooks; literate computing

ACM Classification Keywords
H.5.m [Information interfaces and presentation (e.g., HCI)]: Miscellaneous

Introduction
Given the wide uptake of interactive notebooks such as Jupyter Notebook[1], the software paradigm of literate computing have in recent years gained significant popularity. According to Fernando Perez, one of the inventors of Jupyter Notebook, literate computing is the act of weaving "a narrative directly into a live computation, interleaving text with code and results to construct a complete piece that relies equally on the textual explanations and the com-

[1] jupyter.org

putational components."[2] Particularly data scientists have adopted tools such as Jupyter Notebook or more recently Observable Notebook[3] to create documents that embed executable program code for e.g. live numerical analysis, statistical modeling, and data exploration.

Literate computing has been promoted as a catalyst for open science [7]. It enables the creation of scholarly documents that include executable computations with their source code interwoven with text. A prominent use of Jupyter notebooks, for example, is the use by researchers from the LIGO observatory to disseminate the results on gravitational waves[4].

Jupyter Notebook is the most widespread and popular literate computing environment. It relies on a web-based frontend that communicates with a Python backend, so-called kernels, capable of executing code in various languages (e.g., Python, JavaScript, Haskell, Julia). After code execution, the backend returns the result to the frontend. Users can share Jupyter notebooks via GitHub or through exports as "ipynb" files. Google's *Colaboratory*[5], combines Jupyter Notebook with Google's real-time collaborative editing as, for example, used in Google Docs. Observable notebooks are JavaScript-based, and all execution happens exclusively in the browser. They are geared towards using modern web frameworks such as D3[6] for visualization. Notebooks made with Observable can be shared with others and forked to create personal copies. An extreme example of literate computing happens in Dynamicland[7] by Bret Victor and colleagues. In Dynamicland, programs (conceptually) reside on printed paper, which is tracked in space and projected on through arrays of projectors and cameras.

[2]http://blog.fperez.org/2013/04/literate-computing-and-computational.html
[3]observablehq.com
[4]https://losc.ligo.org/s/events/GW150914/GW150914_tutorial.html
[5]http://colab.research.google.com
[6]http://d3js.org

Literate computing is also appealing for human-computer interaction standpoint. The distinction between "development and use" of software is blurred, programming in a document allows transforming the document into something that typically is not associated with "a document", yet the structure of the software artifact as a document provides a nice set of familiar affordances of sections and paragraphs in word processing.

In our research group, we have been experimenting with alternative *applications* that could benefit from literate computing. We have developed Codestrates [6], a web-based literate computing environment built on top of Webstrates [3]. With Codestrates, we push the literate computing approach of mixing code and prose beyond the state-of-the-art by allowing users to create and personalize documents, customize the documents' tools, and even reprogram them to fit a particular purpose. We generalize the notion of literate computing to document-centric software where the code lives side by side with other web-based content. In addition to literate computing in the classical sense of Knuth [4] where users create documents with embedded computation, it enables the creation of usable applications where the code at any point can be inspected and changed. By leveraging Webstrates, Codestrates inherently supports real-time collaboration on editing the document contents (e.g., as in Google Colaboratory), but also beyond in collaboratively changing the appearance and interactions of applications built with Codestrates.

[7]https://dynamicland.org

Experiences with Codestrates

Our experiences with using Codestrates over the last one-and-a-half years is that a literate computing environment has the potential to change the way we think and make creative decisions when working with digital artifacts.

Document-based content creation

Interaction in Codestrates is *document-based* rather than *application-based*. Each codestrate is a document that contains functionality to write text and code, apply styles, and store data. These building blocks, so-called paragraphs, empower users to customize documents with additional functionality as the needs arise—in contrast to monolithic applications with their pre-defined and fixed functionality. For example, a user can start with a basic codestrate and use it, e.g., for note-taking. Then, when needs change, they can gradually extend the codestrate with functionality—either by writing code or by importing functionality other users have made.

This "document-based" interaction has changed our own thinking when working with documents from "I need to open Microsoft PowerPoint to create slides for a lecture" to "let's create content for the lecture first and then later transform it to slides."

Sharing functionality

Gradually extending a document with new functionality can result in functionality that might be useful in other cases. For example, a countdown timer initially implemented to notify students about the remaining time during group work in a lecture can also be used for writing with Pomodoro Technique[8].

[8]Pomodoro is a time management method, which uses a timer to break down work into several intervals.

We have realized that sharing functionality between codestrates is essential if we want to avoid the "all-in-one codestrate suitable for every purpose," where all past, present, and future functionality will be part of every basic codestrate. This, of course, would also lead to a user interface cluttered with many buttons and menu items that give access to this functionality even though the user might never need it.

To facilitate easy sharing of tools we have introduced—what we call—*package management* [2][9]. Any functionality of a codestrate can be turned into a package and pushed to a package repository, which is also just a codestrate. We have a designated codestrate in our research group to which we push new functionality (i.e., tools, codestrate extensions). All group members that have access to our package codestrate can pull functionality from it. This allows for reuse of the same functionality in different codestrates and at the same time sharing it with others.

We have developed myriads of tools and functionality to support different tasks ranging from free-hand drawing, creating presentations and coding exercises, making visualizations, connecting to IoT devices (e.g., Arduino Uno), and promote remote collaboration including video communication. We have experienced an excitement in writing new functionality as packages and share it with others as well as to try out new packages written by others.

Types of "applications"

A codestrate can be created as a prototype of another, and, e.g., a combination of PDF annotation functionality and a notetaking tool serves as excellent paper reviewing codestrate prototype.

[9]Implemented by Marcel Borowski, University of Konstanz

We have created codestrates for making programming assignments for our programming classes, including scaffolding for the instructor to write “code tests” to give instant feedback to the students while solving the exercises. Each student creates a copy of an exercise codestrate, and when finished sends the link to the teaching assistant for comments.

When making programming exercises it can be difficult to assess whether they hit the right level for the students. We can add survey functionality to an exercise codestrate to get additional feedback from the students (e.g., multiple-choice or open-ended questions). Their survey response is then stored in the codestrate and together with their solution to the exercise.

With our approach to literate computing, the user does neither have to anticipate the outcome or product nor choose an *appropriate* application, but can begin working, e.g., on coding exercises and later decide that a survey is needed, and add that. This is a workflow that is very different from the traditional application-based model where you have to decide in advance what type of outcome you will have to choose an application for making it.

Interest in Workshop

With this position paper, we are stating our interest in discussing the challenge of creating “interactive digital environments that are flexible enough to support appropriation by end users.”

We envision software tools that stand in a dialectical relationship or partnership with the user; software tools that mature and improve with users’ skills and vice versa. We have taken a small step in this direction with Codestrates and its the package management functionality. It allows users to reconfigure their software—without necessarily having to know programming or engage in programming.

We are interested in discussing how the approach to software that we have taken align with the principles of instrumental interaction [1] and co-adaptation [5].

REFERENCES

1. Michel Beaudouin-Lafon. 2000. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proc. CHI ’00*. ACM, 446–453.
2. Marcel Borowski, Roman Rädle, and Clemens Nylandsted Klokmose. 2018. Codestrates Packages: An Alternative to "One-Size-Fits-All" Software. In *Proc. CHI ’18 Extended Abstracts (CHI EA ’18)*. ACM, New York, NY, USA.
3. 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*. ACM, 280–290.
4. Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.
5. Wendy E Mackay. 2000. Responding to cognitive overload: Co-adaptation between users and technology. *Intellectica* 30, 1 (2000), 177–193.
6. 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*. ACM, 715–725.
7. Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature News* 515, 7525 (2014), 151.