Codestrate Packages: An Alternative to “One-Size-Fits-All” Software

Marcel Borowski
Human-Computer Interaction Group
University of Konstanz
Konstanz, 78457, Germany
marcel.borowski@uni.kn

Roman Rädle
School of Communication and Culture
Aarhus University
Aarhus N, 8200, Denmark
roman.raedle@cc.au.dk

Clemens N. Klokmose
School of Communication and Culture
Aarhus University
Aarhus N, 8200, Denmark
clemens@cavi.au.dk

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

Copyright held by the owner/author(s).
CHI'18 Extended Abstracts, April 21–26, 2018, Montreal, QC, Canada
ACM 978-1-4503-5621-3/18/04.
https://doi.org/10.1145/3170427.3188563

Abstract
We present *Codestrate Packages*, a package-based system to create extensible software within Codestrates. Codestrate Packages turns content creation from an application-centric model into a document-centric model. Codestrate Packages no longer restrict users to the feature set of the application. Instead *packages* allow users to add new features to their documents while already working on them. They can match the features to their current task at hand. Supporting the reprogrammable nature of Codestrates, new features can also be implemented by users themselves and shared with other people without having to leave the document. We illustrate the application of Codestrate Packages in an example scenario and present its technical concepts. We plan to conduct multiple user studies to investigate the benefits and barriers of Codestrate Packages' document-centric approach.

Author Keywords
Reprogrammable systems; extensible systems; package management; document-centric.

ACM Classification Keywords
H.5.2 [Information interfaces and presentation (e.g., HCI)]: User Interfaces

Introduction

The creation of digital content is dominated by an application-centric model. When we want to produce content on a computer, we first have to choose a particular application to do it in. For example, we use Microsoft Word to write text, Microsoft Excel to crunch numbers and generate graphs, Adobe Illustrator to create figures and illustrations, and Apple iBooks to read documents.

By doing this, we *a priori*—and unwittingly—limit ourselves in creativity and expressiveness determined by the tools available in an particular application. A Microsoft Word document with notes on the structure of a website does not easily become the final outcome, an interactive website. An Apple Keynote presentation of an initial research idea cannot turn into the final research article, and given the current application-based paradigm it seems absurd to even think that it would. Today, producing content involves juggling multiple applications. Besides adding mental and organizational overhead, the transfer of content from one application to another often leads to a loss of information and interactivity: Exporting a vector graphic to a raster graphic will result in degradation of quality when applying 2D transformations such as scaling or rotation, and a chart generated from a data set imported onto a slide in a presentation will loose its connection to the source data and whatever interaction capabilities it may have had in the charting application. While the application-centric model has worked for decades, it poses several challenges for human-computer interaction.

Unused functionality: Users have to own multiple applications. And even though they only use a fraction of the functionality, they have to pay for all of it.

One-size-fits-all: Users' skills, competencies, and preferences are diverse. Modern software, however, is generic and designed to cater a broad audience. This generality combined with the unused functionality results in applications bloated with functionality making it difficult for the novice to learn. For example, each Microsoft Office application has a tool-set to process images—yet every tool-set is slightly different.

Content and functionality silos: Modern applications silo functionality and content types, which makes it difficult to mix content without a degradation of quality.

Extension sharing: When a user requires functionality that goes beyond what a particular application provides, most applications allow for installing extensions. However, these extensions are installed in the application, and sharing a document created with the extension requires collaborators to have the same extensions installed.

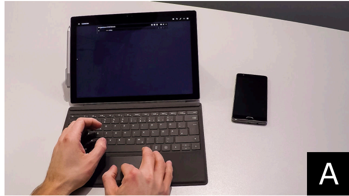
Extension creation: Writing extensions will typically require installing and setting up a particular development environment, and is, therefore, not a practice that is adopted by even users literate in programming.

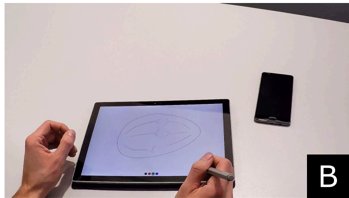
In this paper, we propose a document-centric model for creating digital content where users can (i) add new functionality as the need arises, (ii) implement custom functionality within the document, and (iii) share these implementations across documents and with other users. We present Codestrate Packages, as an evolution of Codestrates [9], a literate computing environment built on top of Webstrates [5].

Scenario of Use

Consider the following scenario of Daniel, a persona created to explain the usage and functionality of Codestrate Packages[1] (see Figure 1):

[1] Illustrated in the accompanying video.





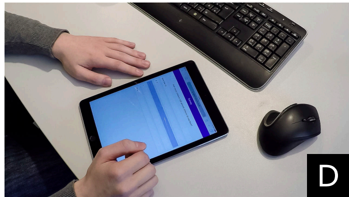


Figure 1: Example scenario:
A: The user uses the *text tools* package for rich-text editing.
B: The user uses the *drawing* package to add a sketch.
C: The user presents the converted notes and the drawing using the *presentations* package.
D: A colleague answers the survey on his tablet.

Daniel, a bioinformatician, is writing a research article on *cell mitosis*[2]. He creates a blank document and starts typing in notes while he reads up on the literature. Quickly, he discovers that he needs more sophisticated text formatting and installs a package for rich-text editing. Moments later, when reading about *nondisjunction*[3], he realizes that he needs to draw a sketch to understand it properly. Daniel installs a drawing package, opens the document on his tablet that has a pen, and draws the sketch. The following day Daniel wants to present his work to his colleagues. He installs a presentation package in the document and turns several paragraphs into slides so he can present directly from his document on cell mitosis. As part of the article, Daniel wants to make a short survey of the history of genetic diseases. He is familiar with JavaScript and implements a small survey-tool directly in his document. This tool could also be useful for his colleagues, so Daniel publishes it to their shared package repository. To gather the data, Daniel sends a link to the document to his colleagues and friends, where it is locked to only display the survey. A few days later, Daniel wants to visualize the results of the survey. He installs a plotting package and extends his survey package to plot the results as well. After having written most parts of the article, Daniel wants to discuss some of them with Jim. He installs collaboration packages, which allow him to invite Jim to his document and discuss the text using video communication directly within the document. To finalize the article, Daniel installs a package that displays the article in a format optimized for reading and publishes the link to his article.

Related Work

Multiple authors have criticized application-based computing for providing monolithic software packages with too many features for the ordinary users, and how they are isolated silos of functionality (e.g., Norman [7], Raskin [10] and more recently Nouwens and Klokmose [8]). Apple's OpenDoc or Microsoft's OLE were commercial attempts to break the application silos through a component based approach to create compound documents. Instrumental Interaction [3] is a radically different software paradigm where tools are proposed to be decoupled from the domain objects they work on, and where users can mix and match tools across different types of domain objects and across different devices [4]. In activity-based computing [2], an activity layer is built on top of regular applications to enable resuming and migrating activities spanning multiple applications. Haystack [1] is an environment for personal information management that allows for creating compound documents of heterogeneous data with multiple different types of views on the data. Finally, interactive or computational notebooks (such as Jupyter [6]) allow users to mix prose and executable code in a literate computing fashion—in effect an environment that allows for changing the nature of the document through programming.

Shareable Dynamic Media

Codestrate Packages is part of a continued effort to realize the vision of Shareable Dynamic Media [5] where software is inherently *shareable* between people, *distributable* across heterogeneous devices, and *malleable* to reprogram and reconfigure it. Klokmose et al. [5] propose to abandon the traditional distinction between applications and documents, and instead build software that is based on *information substrates*, or *substrates* for short. Substrates are software artifacts that can act as both application/tool or document/object depending on their use.

[2] *Mitosis* is a process inside the nucleus during cell division.
[3] A *nondisjunction* is an error that can happen during cell division.

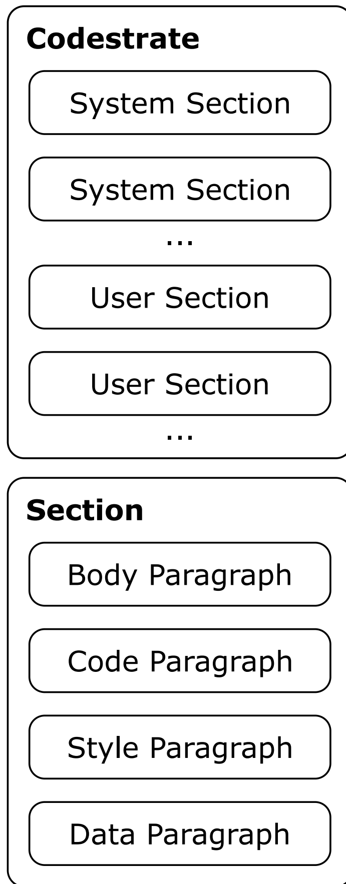


Figure 2: A schematic overview of a codestrate and a section.

Webstrates
Webstrates (web + substrates) [5] is a prototype implementation of shareable dynamic media. It is a web-based platform that introduces a simple but powerful change to how the web otherwise operates. In Webstrates every webpage, called a webstrate, is a collaboratively malleable object. Changes to the document object model (DOM) of a webstrate are persisted to a server, and synchronized in real-time to all other clients of the same page. This includes changes to inlined scripts and style sheets.

Codestrates
A codestrate [9] is a webstrate that includes a literate computing environment (inspired by interactive notebooks such as Jupyter [6]). Codestrates pushes the literate computing approach of mixing code and prose beyond the state-of-the-art by making documents and their tools reprogrammable and personalizable from within. Codestrates inherently supports real-time collaboration and allows for not only creating documents with embedded computation but also usable applications where the code at any point can be inspected and changed.

A codestrate is structured in *sections* consisting of *paragraphs* (see Figure 2). A paragraph can be of the type *body*, *code*, *style*, and *data*. A body paragraph contains regular web content. A user can by default input (rich) text into them or inspect and modify the HTML of a body paragraph directly. Code paragraphs contain executable JavaScript code. They can be executed manually or set to be executed on page load. A style paragraph contains cascading style sheet (CSS) rules, and finally a data paragraph can contain data in the JavaScript Object Notation (JSON) format.

To create an application using Codestrates one would create a body paragraph with the user interface, a code paragraph for the behaviour and a style paragraph for the presentation. The body paragraph can then be put into full screen, creating an app like experience. For a simple application (like for example a to-do list) the application state could be stored as HTML in the body paragraph.

Codestrate Packages

Codestrate Packages adds package management functionality to Codestrates inspired by package management systems such as the Node.js package manager (NPM)[4] and extension managers in modern code editors (as for example in Microsoft's Visual Studio Code[5]). Codestrate Packages is integrated into Codestrates and the code is available on GitHub[6].

The two main parts of Codestrate Packages are the *packages* themselves, which can be added or removed while working in a codestrate, and *repositories*, other codestrates from which packages are pulled or pushed.

Packages
A package in Codestrate Packages extends a codestrate with functionality. For example, the *drawing* package allows a user to turn a paragraph into a drawing canvas. Packages are a new section type in Codestrates.

As seen in Figure 3 a package consists of a documentation of the functionality, properties, which provide metadata about a package, the implementation of the functionality itself and assets like images or JavaScript files, which can be uploaded to a codestrate.

Any user section of a codestrate can be converted into a package after it's implementation. Existing features of

[4] https://www.npmjs.com (last accessed February 19th, 2018)
[5] https://code.visualstudio.com (last accessed February 19th, 2018)
[6] https://github.com/Webstrates/Codestrates (last accessed February 19th, 2018)

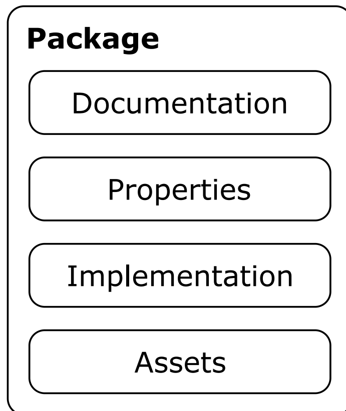


Figure 3: A schematic overview of the structure of a codestrate package.

Codestrates can thereby easily be converted into codestrate packages.

Package Repository
Every codestrate *is* a repository with no or multiple *installed* packages. Each codestrate can pull packages from or push packages to other codestrates; also allowing for a strategy to have dedicated repository codestrates.

When pulling or pushing packages, the repository is being transcluded into the codestrate using an *iframe*. The part of the DOM, which embodies the package, and the assets are then copied from the codestrate to the repository (pushing) or vice versa (pulling).

Discussion and Future Work

Incompatibility between packages
The possibility to mix and match packages poses a technical challenge. It needs to be insured that packages do not interfere with each other. A small number of packages would allow for manual testing. However, this can lead to a complex problem when the amount of packages increases and would require automated testing of all package permutations. Klokmose et al. discuss a similar problem in their VIGO (Views, Instruments, Governors, Objects) architecture [4], describing that the flexibility of the model could prove as a weakness rather than a strength.

Pre-packaged codestrates and package groups
Casual users or novices might be overwhelmed by the number of package combinations. We will experiment with two different approaches to lower the threshold for novices: pre-packaged codestrates and package groups.

Webstrates supports creating new webstrates through prototyping. Therefore, it is possible to create pre-packaged codestrates. A pre-packaged codestrate has "pre-installed" packages, functioning as a template for a specific task. Users, then, can create copies from these templates. For example a *writing* template, which already includes the *text tools*, *light theme* and *word count* packages. When the task changes over time, the feature set of these prototypes could still be extended or reduced like in any other codestrate using Codestrate Packages.

Currently, a user has to individually select packages from a list of packages to add them to a codestrate. We will extend Codestrate Packages by providing groups, to which packages can be assigned to. Package groups could then be installed at once (e.g., all collaboration packages), without the need to individually pick them from the list.

Interaction with software
As described in the introduction, an application-centric model restricts users in that it forces them to choose the application first. By using a system like Codestrate Packages users can start tasks without knowing what exactly the outcome of their task should be (a text-document, presentation, drawing). This gives users more creative freedom, as the threshold to add a package and for example converting notes to slides is lower than copying notes from one application into another to create a presentation.

Beyond that, also the restriction to choose an operating system or device beforehand is lifted. Users can start on any device and seamlessly switch to another. This change of thinking already started by the emergence of online word-processors but would be enhanced even further by Codestrate Packages.

Evaluation
After "dog-fooding" Codestrate Packages within our own research groups, we will conduct multiple user studies to explore the benefits and barriers of a document-centric model

as implemented in Codestrate Packages. We will first focus on using Codestrate Packages for education. Therefore, our initial study will take place at a high-school. Teachers will use Codestrate Packages in different subjects ranging from sciences to humanities. They will use it for several weeks during frontal lecture and for assignments. Pupils use Codestrate Packages for note-taking and solving their assignments. Latter will include both, solving assignments individually and collaboratively in groups.

The study will give insights into the use of modular and document-centric software, e.g., it will reveal typical combinations of packages, but eventually also reveal idiosyncratic ways of use of software that was previously impossible with application-centric software.

Conclusion

Codestrate Packages extends the vision of shareable dynamic media as realized through Webstrates and Codestrates by enabling the creation of digital content where functionality can be added or removed as needs arise. Extensions can easily be shared as *packages* through a shared *repository* which itself is a codestrate, and extensions can be directly created from within a codestrate. Hereby Codestrate Packages presents an alternative to the traditional application silos. Multiple user studies are planned to further investigate the benefits and barriers of Codestrate Packages and a document-centric software model.

REFERENCES

1. Eytan Adar, David Karger, and Lynn Andrea Stein. 1999. Haystack: Per-user Information Environments. In *Proc. CIKM '99*. 413–422.
2. Jakob E. Bardram. 2005. Activity-based computing: support for mobility and collaboration in ubiquitous computing. *Personal and Ubiquitous Computing* (2005), 312–322.
3. Michel Beaudouin-Lafon. 2000. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In *Proc. ACM CHI '00*. 449–453.
4. Clemens N. Klokmose and Michel Beaudouin-Lafon. 2009. VIGO: Instrumental Interaction in Multi-Surface Environments. In *Proc. ACM CHI '09*. 869–878.
5. Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proc. ACM UIST '15*. 280–290.
6. Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, and Jonathan Frederic 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.
7. Donald A. Norman. 1998. *The Invisible Computer*. MIT Press.
8. Midas Nouwens and Clemens N. Klokmose. 2018. The Application and Its Consequences for Non-Standard Knowledge Work. In *Proc. ACM CHI '18*.
9. Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. 2017. Codestrates: Literate Computing with Webstrates. In *Proc. ACM UIST '17*. 715–725.
10. Jef Raskin. 2000. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley Professional.