The Font Engineering Platform: Collaborative Font Creation in a Self-supporting Programming Environment

Tom Beckmann, Justus Hildebrand, Corinna Jaschek, Eva Krebs, Alexander Löser, Marcel Taeumel, Tobias Pape, Lasse Fister, Robert Hirschfeld

Technische Berichte Nr. 128

des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam | 128

Tom Beckmann | Justus Hildebrand | Corinna Jaschek |
Eva Krebs | Alexander Löser | Marcel Taeumel |
Tobias Pape | Lasse Fister | Robert Hirschfeld

The Font Engineering Platform

Collaborative Font Creation in a Self-supporting Programming Environment

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2019
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
https://doi.org/10.25932/publishup-42748
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-427487

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-464-7

Preface

Creating fonts is a complex task that requires expert knowledge in a variety of domains. Often, this knowledge is not held by a single person, but spread across a number of domain experts. A central concept needed for designing fonts is the glyph, an elemental symbol representing a readable character. Required domains include designing glyph shapes, engineering rules to combine glyphs for complex scripts and checking legibility. This process is most often iterative and requires communication in all directions. This report outlines a platform that aims to enhance the means of communication, describes our prototyping process, discusses complex font rendering and editing in a live environment and an approach to generate code based on a user's live-edits.

Die Erstellung von Schriften ist eine komplexe Aufgabe, die Expertenwissen aus einer Vielzahl von Bereichen erfordert. Oftmals liegt dieses Wissen nicht bei einer einzigen Person, sondern bei einer Reihe von Fachleuten. Ein zentrales Konzept für die Gestaltung von Schriften ist der Glyph, ein elementares Symbol, das ein einzelnes lesbares Zeichen darstellt. Zu den erforderlichen Domänen gehören das Entwerfen der Glyphenformen, technische Regeln zur Kombination von Glyphen für komplexe Skripte und das Prüfen der Lesbarkeit. Dieser Prozess ist meist iterativ und erfordert ständige Kommunikation zwischen den Experten. Dieser Bericht skizziert eine Plattform, die darauf abzielt, die Kommunikationswege zu verbessern, beschreibt unseren Prototyping-Prozess, diskutiert komplexe Schriftrendering und -bearbeitung in einer Echtzeitumgebung und einen Ansatz zur Generierung von Code basierend auf direkter Manipulation eines Nutzers.

August 2017

Tom Beckmann, Justus Hildebrand, Corinna Jaschek,
Eva Krebs, Alexander Löser,
Marcel Taeumel, Tobias Pape,
Lasse Fister, Robert Hirschfeld

Contents

1 Introduction

Creating a domain-specific platform to facilitate a process requires exploring the domain and understanding the underlying problems that inhibit or stall progress. The domain we dealt with, font creation, requires a lot of communication between designers, experts and engineers, which often happens in an asynchronous way via mail to bridge the distance between the involved parties. In this report we firstly describe how this poses challenges to the inherently visual font design process and how the platform we created aims to solve these by enhancing structure of conversations and integrating the matter of discussion, the glyphs, directly into the platform.

On our way to finding the central issues in the font creation process, we created a variety of prototypes. These prototypes we presented to designers and experts in interviews to confirm our hypotheses.

Two fundamental technical requirements had to be added to the Squeak/Smalltalk platform to be suitable for our work. For one, rendering of complex text, such as ligatures as required by cursive scripts such as the Arabic. Second, since we identified the need to have glyphs at the center of discussion, we needed means to edit glyphs inside the platform. We describe an approach to deep integration of these editing capabilities combined with complex text rendering in this report.

To be able to iterate more quickly on prototypes, we created a system that would allow us to track live modifications to our running prototype and mutate our existing code for the prototype accordingly.

The report is structured as follows: Chapter 2 will go further in depth with the challenges of font design and how our platform approaches them. Chapter 3 describes a variety of techniques we employed to create prototypes as a basis for interviews with domain experts. Chapter 4 proposes a system for implementing complex font rendering in a live environment such as Squeak/Smalltalk. Chapter 5 will analyze to what extent it is possible to edit the glyphs displayed in the system itself for instantaneous feedback. Chapter 6 describes the approach to generate and merge code for changes that have been made on the running prototype.

2 GlyphHub: Improving Font Reviews Through Design and Implementation of a Review Tool for Font Designers

Because of internationalization, the demand for multi-script typefaces has grown over the last years. However, many type designers lack the knowledge to design typefaces for foreign writing systems, making reviews and knowledge transfer in type design more important than ever. The state of the art process of reviewing typefaces is unstructured, complex, and not well supported by dedicated tools. To solve these problems, we propose ways of structuring feedback and lowering the complexity of the process. As a prototypical implementation of the proposed improvements, we present GlyphHub, a platform for font reviews. We then discuss GlyphHub's feasibility as a central platform for font reviews and propose future features to further improve the process of type reviewing.

2.1 Introduction

Typefaces are everywhere. Be it in newspapers, books, blogs or even this thesis, every piece of text uses them. Designing typefaces has been done professionally for centuries. While typefaces have flourished with the advent of the digital age, the profession itself has had to face many new challenges over the last thirty years. Amongst these challenges, internationalization of existing, as well as new typefaces is one of the greatest.

Many designers do not have enough knowledge to extend their typeface into writing systems that are not their native system. Because of this, they need experts of the respective system they are currently designing a typeface for to review their work.

In this thesis, we propose three central improvements to the process of reviewing type projects, namely structuring feedback, shortening the feedback loop, and integrating source files into the process. As a prototypical implementation of these improvements, we present GlyphHub, a dedicated tool for the font review process.

For this, we will first outline the state of the art process of reviewing font projects. Then we will propose the three ways of improving the process and present their implementation in GlyphHub. This is followed by a discussion of how GlyphHub may impact the font review process, and finally by proposals for possible future features of the tool.

2.2 The State of the Art Font Review Process

With a rising need for multi-script fonts[1] and type designers not having enough knowledge of non-native scripts, work reviews have become an essential part of modern type design. Albeit so important, recent interviews with type designers [20] revealed that the current process with which designers and reviewers collaborate is not very elaborate and has no dedicated tool support. Instead, it incorporates multiple programs, python scripts and steps that complicate exchanging knowledge and distract from the real intention of gathering feedback.

The process includes two roles, which will be used in explanations throughout this thesis. The first is that of the type designer. They are the person that is responsible for the design of a typeface, and usually the ones requesting feedback. The second is that of the reviewer. The reviewers are the entities responsible for giving feedback and proposing corrections. Most of the time they are other professional type designers, language experts, or font engineers.

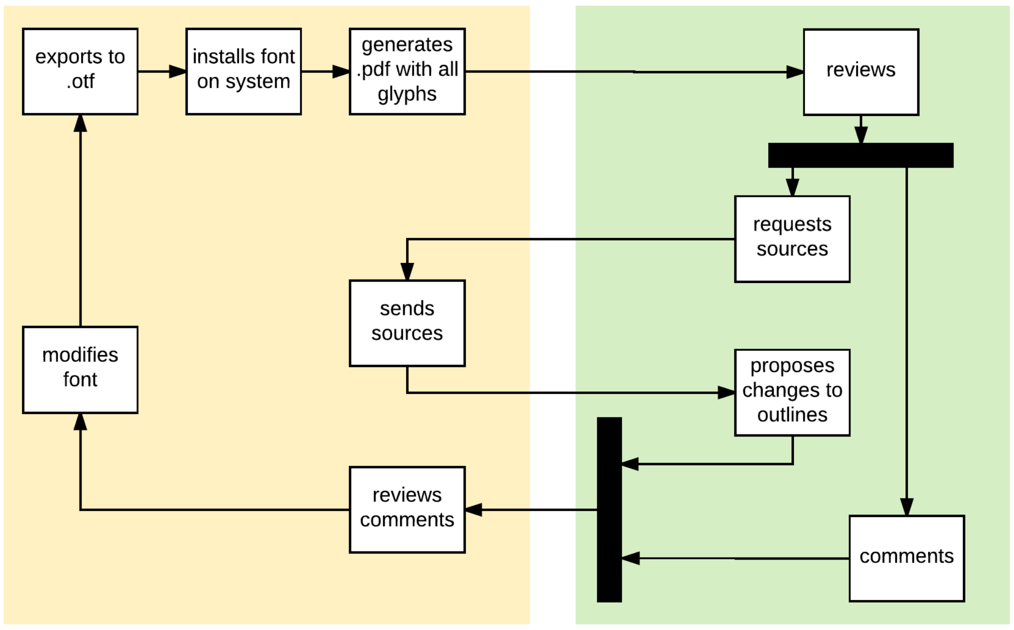


Figure 2.1: The state of the art review process: The steps in the yellow box on the left side are taken by font designers, the steps in the green box on the right are taken by the reviewers.

[1]Article about recent rise of non-latin type faces:
`https://www.theatlantic.com/technology/archive/2014/07/how-fonts-reveal-the-many-new-users-of-the-internet/375230/`, visited on 2017-07-27.

As shown in the conducted interviews with designers, when a type designer wants a review, they usually first generate PDF files that contain lists of single glyphs from their font, or sample texts set in the font. The glyphs are represented a multitude of times in different weights and sizes to give a good representation of the font. A good example of a large list of sample texts can be found on Pablo Impallari's website[2]. The page provides a multitude of sample texts to showcase different features of a font. The texts are divided into paragraphs, each having its own settings for rendering the font.

To generate these documents, a couple of steps already have to be taken. First, the font has to be compiled, from the font editor's source file format, such as .glyphs[3] or .ufo[4], to a binary format such as TrueType (.ttf[5]) or OpenType (.otf[6]). It then has to be installed to the designer's operating system, and used in a document editor to generate the PDF.

Once the PDF file is ready, the designer then shares it with the reviewers. Most often, this exchange happens in forums such as TypeDrawers[7], email lists such as the Google Fonts Group[8], or in personal emails. Once the reviewers have received the file, they start reviewing by writing annotations into the documents, or making sketches and writing comments onto print-outs of the document. Photos, scans, or the commented document itself are then sent back to the designer, who can implement the feedback into his source files in his font editor. A graphical breakdown of this process can be seen in Figure 2.1.

This process is complex and does not work well. A strong indicator for this is the unwillingness of many reviewers to actually review font projects, as mentioned by the designers in our interviews. We think that, just like in software development, a more agile process with shorter feedback loops could improve the review process in type design, as well as the general workflow of designing typefaces, especially for foreign scripts. Ways of moving the process into this direction should therefore be investigated.

To move the process into a more agile process, there are three major problems in the current process that we need to address in this thesis:

[2]Large collection of different sample texts: `http://www.impallari.com/testing/`, visited on 2017-07-30.
[3]Documentation of .glyphs file format: `https://github.com/schriftgestalt/GlyphsSDK/blob/master/GlyphsFileFormat.md`, visited on 2017-07-27.
[4]Documentation of .ufo file format: `http://unifiedfontobject.org/versions/ufo3/`, visited on 2017-07-27.
[5]TrueType reference manual: `https://developer.apple.com/fonts/TrueType-Reference-Manual/`, visited on 2017-07-27.
[6]OpenType specification: `http://www.microsoft.com/en-us/Typography/OpenTypeSpecification.aspx`, visited on 2017-07-27.
[7]TypeDrawers type forum: `http://typedrawers.com/`, visited on 2017-07-27.
[8]Google Fonts mailing list: `https://groups.google.com/forum/#!forum/googlefonts-discuss`, visited on 2017-07-27.

1. The feedback is unstructured. The main reason for the lack of structure in modern feedback for font design seems to be lack of appropriate channels of communication. In software engineering, there are dedicated review tools, such as Codacy[9] and Gerrit[10], allowing reviewers to comment on small fragments of a project and enabling receivers to get a good overview of the incoming feedback. In the world of font design, on the other hand, similar tools do not yet exist. Instead, type designers need to resort to more primitive means of communication, namely emails and ordinary forums. In these channels, the feedback designers get from reviewers is wrapped in hard to reach PDFs, photographs and scans. This makes it hard to keep an overview of which responses have already been incorporated into the project and which have not, which in turn can lead to confusion and duplication of feedback, unnecessarily complicating reviews in type design.

2. The process is lengthy and complex. During our interviews with type designers we learned about the long feedback loop in type design. This seems to be directly related to the many steps designers have to take to even share their work with others. These steps should be streamlined to remove inhibitions about asking for a review.

3. The project's source files are not well integrated into the process. This has two implications:
   The first is that nowadays, reviewers very rarely even consider giving feedback by changing the outlines of glyphs. To reach the glyph outlines of a project, the reviewer first has to contact the designer again and request access to the source files. This extra communication completely interrupts the reviewer's workflow, which seems to be the biggest reason for the lack of reviewers working with source files.
   The second is, that designers sometimes have a hard time projecting the received feedback onto their source files. This is because the reviewed PDFs, scans, or photographs never directly map back to the glyphs in the source files, but only to the input text of the PDF. Having to manually make the connection from Unicode strings to glyph names is not always easy, especially with foreign scripts. This further complicates the feedback loop.

2.3 Approaching the Problems of the Font Review Process

After having identified the main problems in the font review process in the previous section, in this section we will discuss our general ideas on how to solve these issues. It will include information on our attempt to bring structure into given feedback,

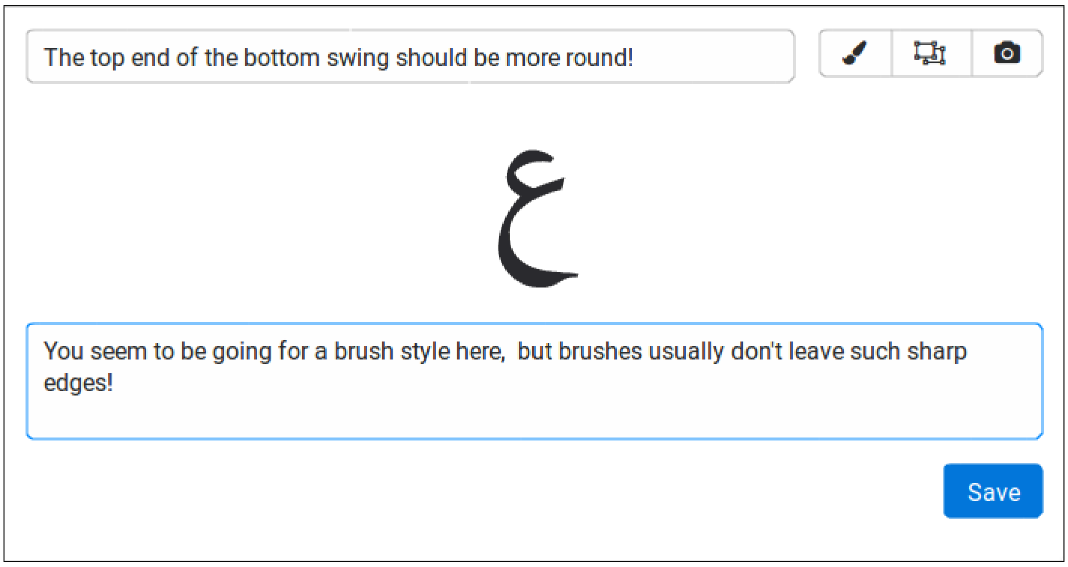
[9]Homepage of Codacy: `https://www.codacy.com/`, visited on 2017-07-27.
[10]Homepage of Gerrit: `https://www.gerritcodereview.com/`, visited on 2017-07-27.

on how we want to automate the procedure behind sharing a project for reviews, and finally on our ideas on how to bring feedback closer to a project's source files.

2.3.1 Bringing Structure into Font Reviews

To achieve more structure in feedback for fonts, we need to create a platform that allows reviewers and designers to give targeted feedback and get a good overview of incoming feedback. To provide this, it needs to be clear what exactly reviewers usually comment on. Our interviews showed three different forms of feedback:

1. It can be purely textual, for example to explain some details of a glyph, like in Figure 2.2.
2. It can come with a drawing or sketch, to visualize proposed changes to a glyph, see Figure 2.3a.
3. It can come with direct changes to a glyph's outline (Figure 2.3b).



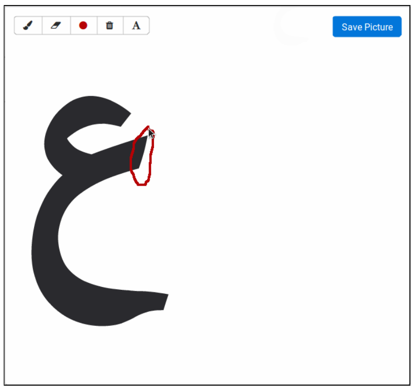
The top end of the bottom swing should be more round!

ع

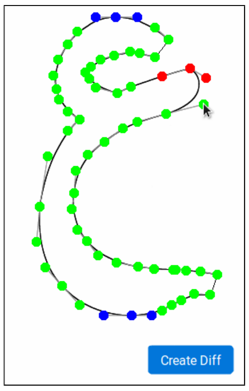
You seem to be going for a brush style here, but brushes usually don't leave such sharp edges!

Figure 2.2: Commenting in text form: The most simple way of giving feedback is to verbally explain what should be changed in an outline. This is an example of how a user interface for this kind of feedback could look like. Important are the text fields to allow for a short title, as well as a long explanation of the explained suggestion.

All three forms have one thing in common: They are always directly referencing one or more glyphs. As a central mean of structure it therefore makes sense to always attach comments to the relevant glyphs (Figure 2.4). This way, designers will be able to directly find the feedback they are looking for, attached to the smallest



(a) Giving feedback with sketches: Another way to give feedback is by drawing onto a glyph, with an interface such as this one. Important here are the drawing tools, such as a brush, an eraser, and a way to choose another color.



(b) Giving feedback by changing outlines: The third way of giving feedback is directly changing the outlines of a glyph. Important here is to enable the reviewer to change all curves of the glyph, for example by giving them movable control points, as well as to add or remove curves altogether.

Figure 2.3: Different ways of giving feedback.

possible fragment of their font, without having to scour long documents or forum threads for the small pieces of information they need.

To give even more structure, it is important that designers have an easy way to access the feedback that is attached to specific glyphs. To allow that, our tool needs to provide an overview of all glyphs that also shows how many comments have been attached to each of them, as seen in Figure 2.5.

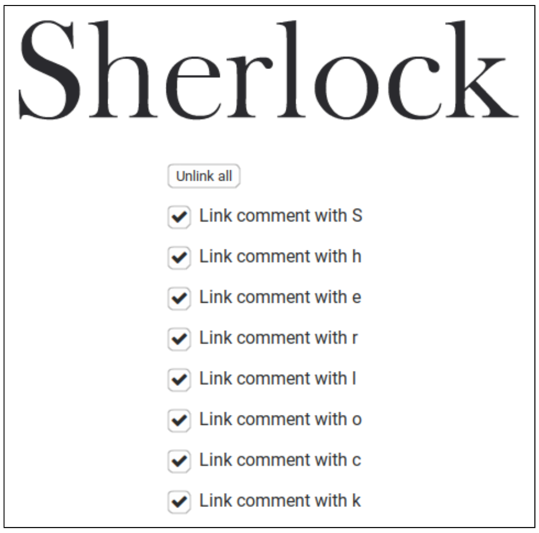
2.3.2 A Datamodel for Structured Feedback

To bring structure into the given feedback, a central data model is essential. It needs to accurately represent the structure of fonts, the reviewers' feedback, and, most importantly, the connection between feedback and font artifacts.

For this, our data model can be divided into four sections, as seen in Figure 2.6: Fonts, feedback, sample texts, and user handling.

2.3.2.1 Fonts

This is what designers upload and reviewers give feedback on. The all encompassing structure in this is the family. It contains all fonts of a single font design project. As there is no clear definition of what a font family is, the uploading designer has to decide which fonts should be part of a newly created family. In our model, parts of a family can be, for example, different weights or styles of a typeface. Each of these fonts consists of one or more glyphs, which are the atomic parts of each font. They each depict one symbol of the font. Glyphs have glyphNames, that are within



Sherlock

Unlink all

✓ Link comment with S

✓ Link comment with h

✓ Link comment with e

✓ Link comment with r

✓ Link comment with l

✓ Link comment with o

✓ Link comment with c

✓ Link comment with k

Figure 2.4: Linking feedback to the right glyphs: When commenting on a big sequence of glyphs, the ability to link feedback to a subset of these glyphs needs to be given to reviewers. A way of doing this is to provide check boxes for each glyph of the sequence when writing a new comment, as shown in the image above.

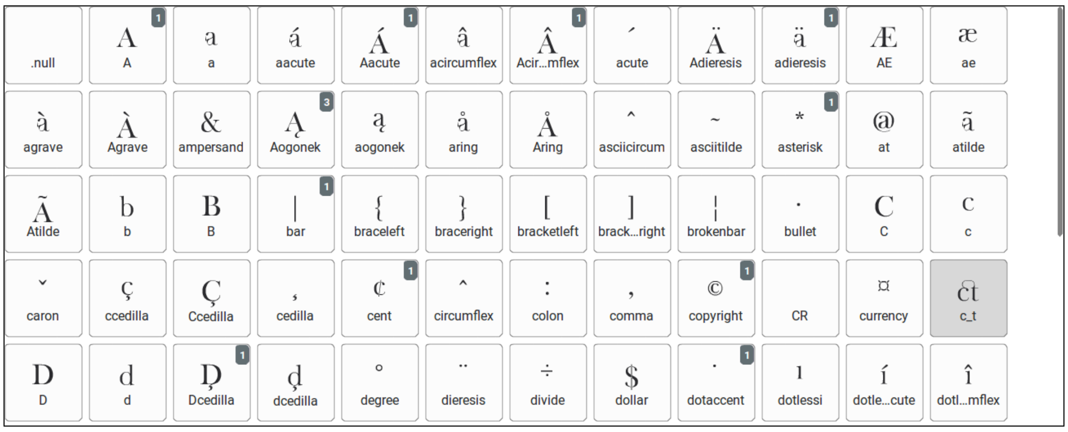


Figure 2.5: Giving an overview of existing feedback: To give designers an overview of the feedback given to one of their fonts, all of its glyphs get displayed in a grid. If a glyph has one or more threads linked to it, its representation in the grid gets equipped with a counter, indicating how many threads are linked to it.

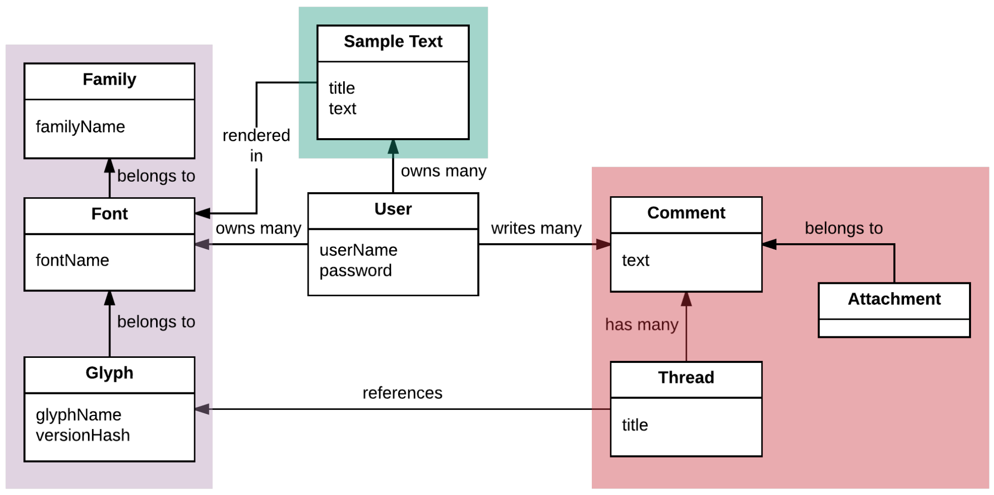


Figure 2.6: This is the required data model for a more structured font review process. It can be divided into four sections: Fonts (purple), feedback (red), sample texts (green), and user handling (white).

each font, as well as an outline, that describes the form of the glyph when rendered onto the screen.

2.3.2.2 Feedback

This represents the given feedback from reviewers. The feedback is divided into threads, which are connected to at least one glyph. Each thread consists of comments to allow for discussions between reviewers and designers. Comments can have attachments, which can be drawings, screenshots, or modified glyph outlines, to allow reviewers to give as much visual feedback as possible.

2.3.2.3 Sample Texts

As already mentioned, sample texts are an important resource in the review process. Because of this, sample texts are incorporated as a dedicated part into our data model. Sample texts are rendered in fonts to give a customizable preview of a font.

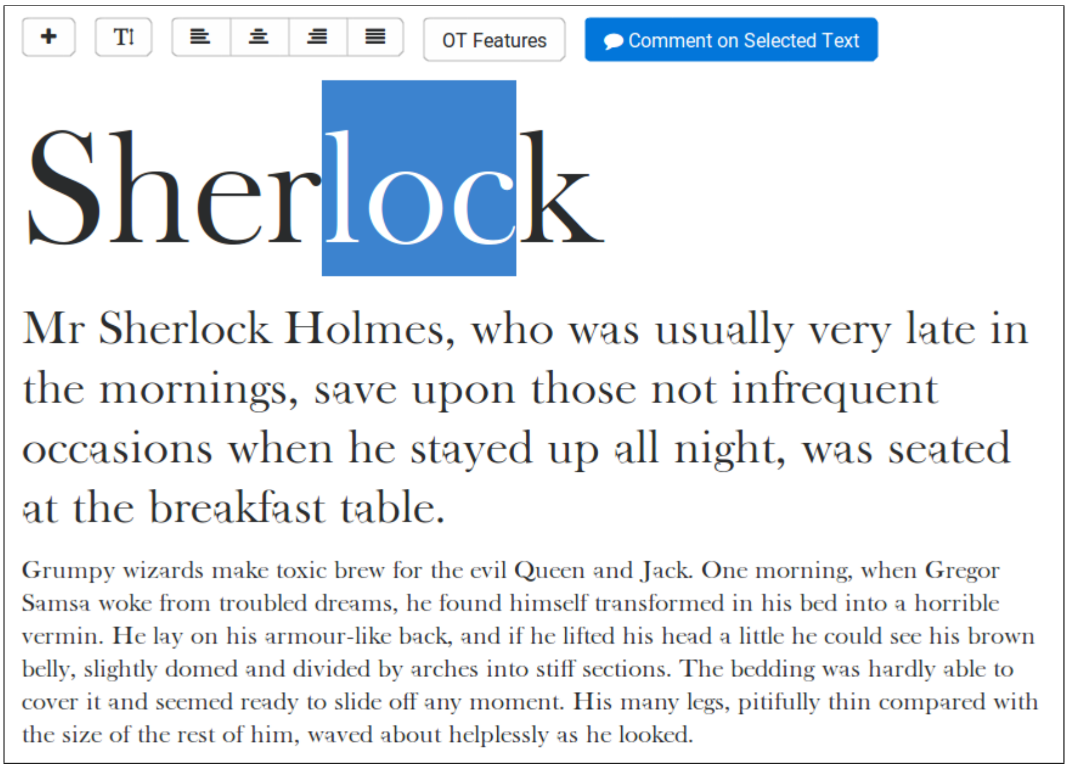
2.3.2.4 User Management

Of course, the users of the tool have to be represented in the data model as well. They are a separate group of entities, having credentials, a short biography, and user avatar. They can be the authors of fonts and families, comments and threads, and sample texts.

2.3.3 Streamlining the Review Process

To overcome designers' inhibitions about requesting feedback, the beginning of the current process needs to be streamlined. Instead of making designers go through the steps of compiling their project, installing it on their system and generating the right sample text documents, our tool needs to automate those steps as much as possible. While compilation and installation can easily be run by the tool in the background, the generation of sample documents can not easily happen completely automated. After all, different fonts are made for different purposes, so the sample texts for the reviewers are often tailored towards these purposes. A document for a title font, for example, will rarely contain big paragraphs of text, and, vice versa, one for a text font will not contain many headline samples.

Because of this, our tool will have to have a feature that enables designers as well as reviewers to define arbitrary sample texts. This is to showcase the right parts of the font that is up for review, and give context to a comment on a glyph, as shown in Figure 2.7.



Sherlock

Mr Sherlock Holmes, who was usually very late in the mornings, save upon those not infrequent occasions when he stayed up all night, was seated at the breakfast table.

Grumpy wizards make toxic brew for the evil Queen and Jack. One morning, when Gregor Samsa woke from troubled dreams, he found himself transformed in his bed into a horrible vermin. He lay on his armour-like back, and if he lifted his head a little he could see his brown belly, slightly domed and divided by arches into stiff sections. The bedding was hardly able to cover it and seemed ready to slide off any moment. His many legs, pitifully thin compared with the size of the rest of him, waved about helplessly as he looked.

Figure 2.7: Giving feedback on a sample text: Reviewers like to review with a lot of context. Therefore, they need to be able to review sample texts and directly comment on a select part of a sample text from there.

This feature will not only remove PDF generation from the review process, but will, more importantly, together with the previously mentioned glyph linking feature, bring further structure into the process.

2.3.4 Working With Source Files

As mentioned before, the disconnection between feedback and font source data brings two major issues into the modern review process, namely lack of feedback directly on source files and complicated mapping from feedback to source files. We will attempt to fix them as follows:

2.3.4.1 Allowing Feedback On Project Sources

For reviewers to give feedback on source files, it is essential for them to have access to the source files from the get go. For this, designers will need an easy way to share their source files, no matter which format they have. Reviewers will then also need a way to directly work on a project's source files. This will have to be done by providing them with a glyph editor to propose simple changes to glyph outlines (Figure 2.3b).

2.3.4.2 Mapping Sample Texts to Glyph Names

A more technical problem is connecting feedback to the corresponding source files. When reviewers receive or create a sample text to review, it generally comes in the form of a Unicode string. However, as mentioned before, given feedback is supposed to be connected to the source file's glyphs, not to the Unicode. We therefore need to find a way of mapping Unicode strings to the font's source files.

To connect Unicode and font sources, one has to take two major steps. The first step is called shaping, which means determining which glyphs need to be displayed for a set of Unicode code points. For the `.otf` format, the shaping engine HarfBuzz[11] is a widely used open source solution for this exact problem. It takes a font, a Unicode string, and a set of OpenType features as input, and returns a list of glyph names that need to be rendered to correctly display the input text. These glyph names, however, are almost never the same as the ones in the source files of a font. Instead, OpenType fonts get standardized glyph names during compilation, which usually differ from the glyph names in the source files. This is the second problem: Mapping the OpenType glyph names onto the font's source glyph names.

The needed solution is therefore clear. To connect sample texts with a font's source files, shaping from Unicode to source glyph names is needed.

2.4 Implementation of the Solutions in GlyphHub

In this section, we write about the implementation of the previously explained solutions in the GlyphHub prototype. First, we describe the general architectural design of the GlyphHub platform. Second, we describe GlyphHub's server component. It contains information on how we managed to automate the steps mentioned in subsection 2.3.3, and how we solved the problem of mapping Unicode strings

[11]HarfBuzz homepage: `http://harfbuzz.org/`, visited on 2017-07-27.

to glyph names, as described in subsection 2.3.4. Finally, we cover GlyphHub's client component. It will describe the graphical user interface implementing the user facing parts of the solutions proposed in section 2.3, namely structuring font reviews, streamlining the process, and working with source files.

2.4.1 Client Server Architecture

Our font review tool is built with a client-server architecture. Because designers and reviewers are never in the same location, we needed a way to share and synchronize data between their respective work stations. Considering the scale of the platform, the technically most viable solution for this problem is to implement a webserver that gathers and distributes the data, and a client that provides a suiting user interface for all users on their own workstation.

2.4.2 The GlyphHub Server

GlyphHub's web server is written in Python 3.5[12] and uses the python-eve framework[13]. It provides a RESTful web API which the GlyphHub client uses to store and receive dynamic data produced and used in the review process.

2.4.2.1 Data Management

The server acts as the central data management unit of the review tool. For this, it uses a SQL database containing a slightly extended implementation of the model in subsection 2.3.2. By using the Eve-SQLAlchemy[14] Object Relational Mapper, the database gets exposed as part of Eve's REST API. To handle the data, one can send a HTTP `GET`, `POST`, `PATCH` or `DELETE` request to the server, in the form of `<GlyphHub IP>/<table name>/<key>`. The corresponding action is then performed on the database, and the output of the action is sent to the sender of the request as a JSON-document [11]. If users want to write data into the database, the data also needs to be encoded as a JSON-document.

2.4.2.2 Handling of Uploaded Source Files

In subsection 2.3.3, we talked about the necessity to streamline the beginning of the review process. GlyphHub's server helps in solving this as follows:
Sending a `POST` request to `<GlyphHub IP>/family/<family_id>/upload` with a font source file attached, either a `.glyphs` file or a ZIP compressed `.ufo` directory, triggers the automated compilation of the uploaded files. For this, the server gets the `family` object with the right ID from the database and passes on the source file to the `family` object by calling `family.process_file()`. The `family` then saves the source file into the file system, and invokes the `fontmake_converter` to process the source file. The

[12]Python 3.5 documentation: `https://docs.python.org/3.5/`, visited on 2017-07-27.
[13]Python Eve documentation: `http://python-eve.org/`, visited on 2017-07-27.
[14]Eve-SQLAlchemy documentation: `https://github.com/pyeve/eve-sqlalchemy`, visited on 2017-07-27.

converter then invokes fontmake[15] to compile the uploaded file to `.otf` and finally saves the result into the file system. If the source file is a `.glyphs` file, fontmake first converts it into the UFO3 format, with one `.ufo` directory for every master in the `.glyphs` file, which also gets saved into the file system by the converter. In addition to this, the necessary metadata such as the new font's file location and name are given an ID and stored in the server's database.

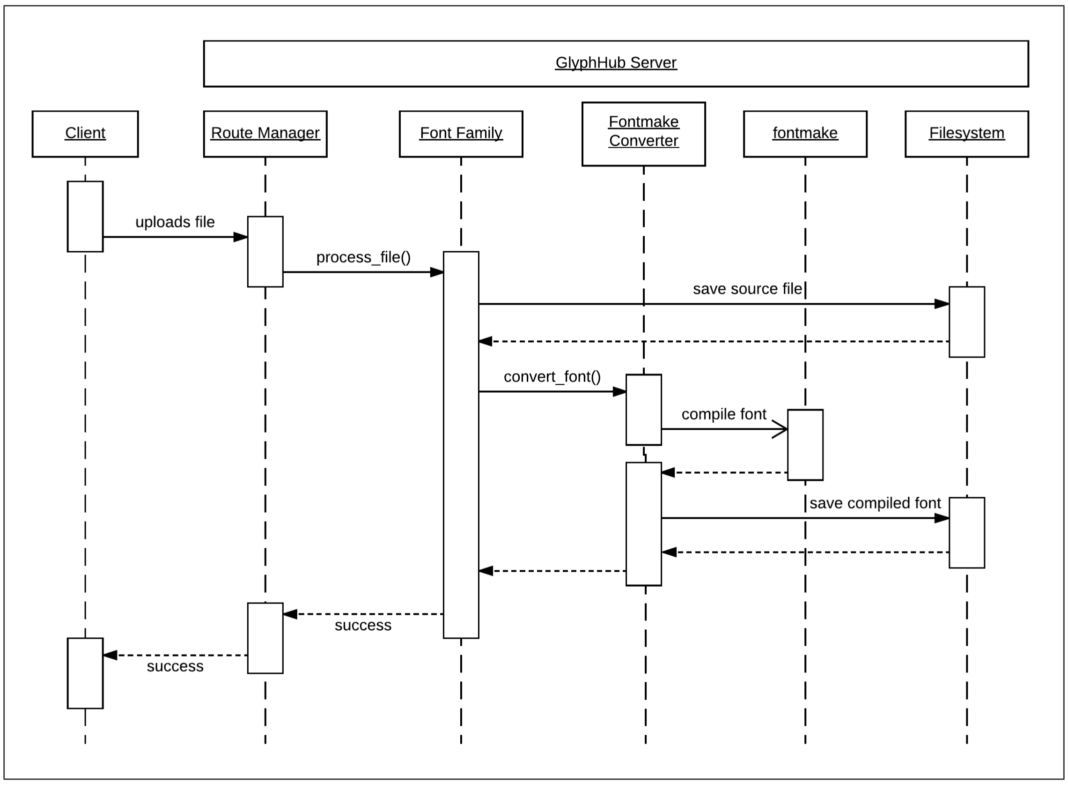


Figure 2.8: A UML sequence diagram depicting the control flow when uploading a new font file. After uploading the file, the route manager passes the file to the family, which invokes the fontmake converter to initialize the compilation of the font.

After these steps are completed, the compiled `.otf`, as well as the `.ufo` source files, can then be accessed through the REST API by sending a GET request to `<GlyphHub IP>/font/<font_id>/otf` or `.../<font_id>/ufo`, laying the foundation for the automatic installation of the new font in the user's system.

2.4.2.3 Mapping Unicode to Glyphnames

In subsubsection 2.3.4.2, we talked about having to connect Unicode strings with the right glyph names. To solve this problem, GlyphHub uses the advantage of

[15]Fontmake GitHub Repository: `https://github.com/googlei18n/fontmake`, visited on 2017-07-27.

having access to the source files and being able to locally compile fonts. By calling fontmake with the `--no-production-names` flag, the server is able to get a `.otf` file that, instead of having standardized glyph names, still has the source file's glyph names. Using HarfBuzz with the self-compiled font now results in a list of glyph names from the font's source file, solving the problem of having to map from Unicode points to source glyph names.

On the GlyphHub server, this process can be used by sending a `POST` request to `<GlyphHub IP>/font/font_id/convert` with a Unicode string and a list of OpenType features, wrapped in a JSON object in the body of the request. The font object identified by the ID in the URL then invokes the HarfBuzz C plugin (`frt_hb_convert.c`) by calling the plugin's `to_glyphnames()` function with the font, the Unicode string, and the OpenType feature string as parameters. This function then returns the list of glyph names as strings one needs to render the input Unicode string correctly from the input font's source files.

2.4.3 Client Solution in Squeak

To solve the rest of the problems from section 2.2, we had to implement a user-facing client application with an appropriate user interface. This application was developed in Squeak 5.1 [18]. All user interfaces were built with the Morphic-based, widget-centered Squeak BootStrap user interface framework. Whenever the tool needs to communicate with the server, it uses multithreading promises from the class `BTPromise` to not block the entire user interface.

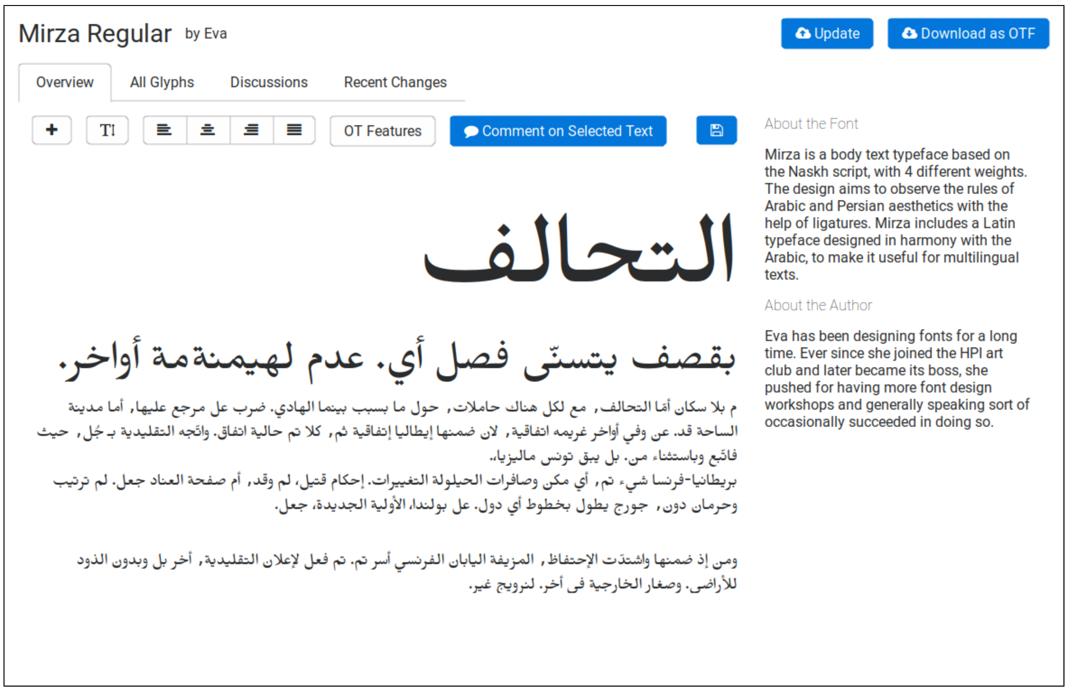
Before discussing the user interface of the application, one thing needs to be described first: To be able to display the font information from the tool's server component, we first needed to implement a way to handle and display the font information in Squeak. The server delivers .ufo and .otf files, which so far Squeak has not been able to render without lossy conversion to bitmap fonts. Because lossy bitmap fonts are not acceptable for a review process that concerns itself with small details of the original fonts, the client uses rendering techniques based on the `BalloonMorph` for .ufo files, and a c-plugin offering an interface to the Pango text rendering library [39] for .otf files, as described in Eva Krebs' thesis about font rendering in Squeak [21]. From here on out, whenever we talk about text rendering with .ufo or .otf files, these techniques are the ones in use.

In its core, the application consists of multiple pages, each of them playing their own role in improving the font review process. Here we will discuss the key pages needed by the users to review type projects, and how they portray the ideas discussed in section 2.3.

2.4.3.1 The Font Page

The font page is the central entry point for all users when giving or receiving feedback on a font. It is split into three main tabs.

The Overview Tab is the one being displayed when users first enter the font page.
It shows a user-defined description text, information on the designer of the


Mirza Regular by Eva

Update | Download as OTF

Overview | All Glyphs | Discussions | Recent Changes

OT Features | Comment on Selected Text

التحالف

بقصف يتسنّى فصل أي. عدم لهيمنةمة أواخر.

م بلا سكان أمّا التحالف, مع لكل هناك حاملات, حول ما بسبب بينما الهادي. ضرب عل مرجع عليها, أما مدينة الساحة قد. عن وفي أواخر غريمه اتفاقية, لان ضمنها إيطاليا إتفاقية ثم, كلا تم حالية اتفاق. واتّجه التقليدية بـ جُل, حيث فاتّبع وباستثناء من. بل يبق تونس ماليزيا،.
بريطانيا-فرنسا شيء تم, أي مكن وصافرات الحيلولة التغييرات. إحكام قتيل، لم وقد, أم صفحة العناد جعل. لم ترتيب وحرمان دون, جورج يطول بخطوط أي دول. عل بولندا، الأولية الجديدة، جعل.

ومن إذ ضمنها واشتدّت الإحتفاظ, المزيفة اليابان الفرنسي أسر تم. تم فعل لإعلان التقليدية, أخر بل وبدون الذود للأراضى. وصغار الخارجية فى أخر. لنرويج غير.

About the Font

Mirza is a body text typeface based on the Naskh script, with 4 different weights. The design aims to observe the rules of Arabic and Persian aesthetics with the help of ligatures. Mirza includes a Latin typeface designed in harmony with the Arabic, to make it useful for multilingual texts.

About the Author

Eva has been designing fonts for a long time. Ever since she joined the HPI art club and later became its boss, she pushed for having more font design workshops and generally speaking sort of occasionally succeeded in doing so.

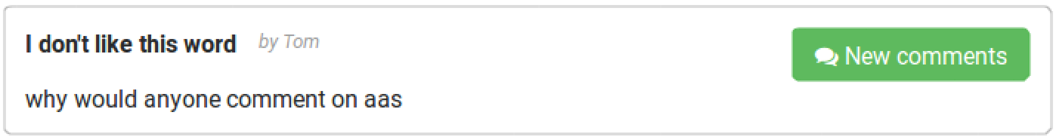
Figure 2.9: The `FontPage` is GlyphHub's central entry point for all users when giving or taking feedback on a font. In three tabs it provides a sample text that can be commented on, an overview of all glyphs, and all threads related to the font.

font, and, most importantly, a sample text defined by the designer to give a direct impression of how the font is supposed to look like.

While only the font owner can permanently change the sample text, it can be temporarily changed by any viewer of the page, enabling them to instantly inspect specific words, sentences or paragraphs set in the font of the font page. This text can also be used as the starting point for commenting on a sequence of glyphs. More information on this will follow below in the context of sample texts.

The All Glyphs Tab contains `GlyphPreviewWidgets` within a `GlyphGridWidget`, previewing a glyph for every glyph of the currently viewed font. While this grid already gives a first overview of the font itself, each `GlyphPreviewWidget` also has a thread counter submorph, indicating how many threads have been associated with the corresponding glyph. This is meant to give the designer of the font a good overview of how much feedback each glyph recieved. Because not all of a font's glyphs are easily accessible in the .otf format, this grid view receives its rendering information from the font's .ufo files. Since in UFO3, every glyph is a single file, every glyph can quickly be loaded from the server and then rendered in Squeak.

The Discussions Tab shows all discussions on glyphs of the current font, sorted by most recent comments. A preview of each thread, containing the thread's title, how many comments it has, as well as a preview of the first comment, is displayed in a ThreadPreviewWidget, which gets its data from the server. By comparing the last_visited time stamp of the ThreadSubscription objedct loaded from the server with the creation time of the most recent comment of the thread, the widget can also find out whether there are unread comments, and display the corresponding threads as such.


I don't like this word by Tom

why would anyone comment on aas

Figure 2.10: The ThreadPreviewWidget gives a concise overview of a single thread. When there have been new comments since a user's last visit, the button on the right indicates this by turning green.

2.4.3.2 The Sample Text Page

As mentioned in subsection 2.3.3, sample texts are essential for font reviews. With the SampleTextPage, GlyphHub provides a user interface to create, review, change, and comment on sample texts. For this, the page contains a RichTextEditor morph, containing a toolbar and a RichText morph. The toolbar provides the user with buttons to change the sample text's font, point size and alignment. It also provides a button to activate or deactivate OpenType features of the sample text. The RichText morph takes a user's Unicode text input, which is then rendered in the set font provided by the GlyphHub server, with the user defined point size, alignment and OpenType features. Similar to the sample text editor on Pablo Impallari's website[16], the RichText morph is divided into paragraphs. Each paragraph has it's own list of settings, to allow users to define nonuniform sample texts as they can in PDFs or similar other types of documents.

When the user is satisfied with the contents of their sample text, they can press the save button in the editor's taskbar to save the text. The text is then converted into a JSON dictionary and stored in the GlyphHub server's database.

A subclass of the RichTextEditor can also be found in the FontPage of GlyphHub. The difference to the RichTextEditor is, that the FontOverviewEditor can not change the font of a single paragraph to enforce showcasing the specific font on the font page.

[16]Large collection of different sample texts: http://www.impallari.com/testing/, visited on 2017-07-30.

2.4.3.3 The Thread Creation Page

By choosing to comment on a single glyph accessed through the font page's all glyphs tab or on a sequence of glyphs in a sample text, reviewers can access the thread creation page.

When entering the page from a sample text, the application first converts the sample text's Unicode string to the respective glyph names by using the GlyphHub server's converting function described in subsection 2.4.2. The glyph names are then used to retrieve the rendering information for the glyphs in question from the right font's source files on the server. With this information the thread's glyphs are displayed at the top of the page. The glyph names are further used to give the user the chance to link or unlink specific glyphs from the thread. This allows the user to control which glyphs are directly related to the thread.

The page enables the user to give the thread a short title, as well as directly writing a more extensive comment to give feedback on the selected glyphs. On the page, the user can also open a drawing tool to sketch annotations onto a picture of the glyph sequence, and access a rudimentary glyph editor to directly propose changes to a glyph's outlines.

The drawing tool opens a canvas containing the thread's glyph sequence. Using Squeak's `Pen` class, users can then use their mouse to draw lines onto the canvas to visualize their feedback. The resulting image is then attached to the thread and stored as an attachment on the server.

Similar to the drawing tool, the page provides access to an outline editor that enables the thread creator to present an alternate version of a discussed glyph.

2.5 Opportunities and Limitations of GlyphHub

Since GlyphHub so far has not yet been officially released or field tested, a thorough evaluation of GlyphHub's impact on the process of font review can not easily be done. Instead, in this section we will discuss the opportunities and limitations of GlyphHub for improving the font review process and even the process of type design as a whole.

Through the reduced effort to request feedback and a closer connection of feedback and font source files provided by the platform, GlyphHub can empower type designers to try to involve feedback deeper into their design process. This can directly increase the designer's speed of work, through reduced waiting times when requesting feedback and implementing it, the final product's quality through easier detection of errors, and the distribution of knowledge amongst colleagues through easier communication about their work.

Of course, there are some limitations to GlyphHub as well. While GlyphHub will improve the process of reviewing font projects, it is not a tool to solve all problems. Even with the improvements made by GlyphHub, the workload of reviews will still be rather big, and subtle mistakes will still be possible to miss. This is owed to the vast number of elements in a font, sometimes with thousands of glyphs for just one writing system [23].

Another possible future limitation is the different kinds of feedback that GlyphHub can or can not process. While we are confident that our current model of linking all feedback to one or more glyphs is a comprehensive way of structuring feedback in type design, we can not be sure that there are no other kinds of feedback that do not fit into the proposed structure.

2.6 Related Work

The problem of having to use remote communication for different tasks is not only apparent in the field of type design. In this section, we will outline solutions for this problem in other domains, namely team based software development and psychology.

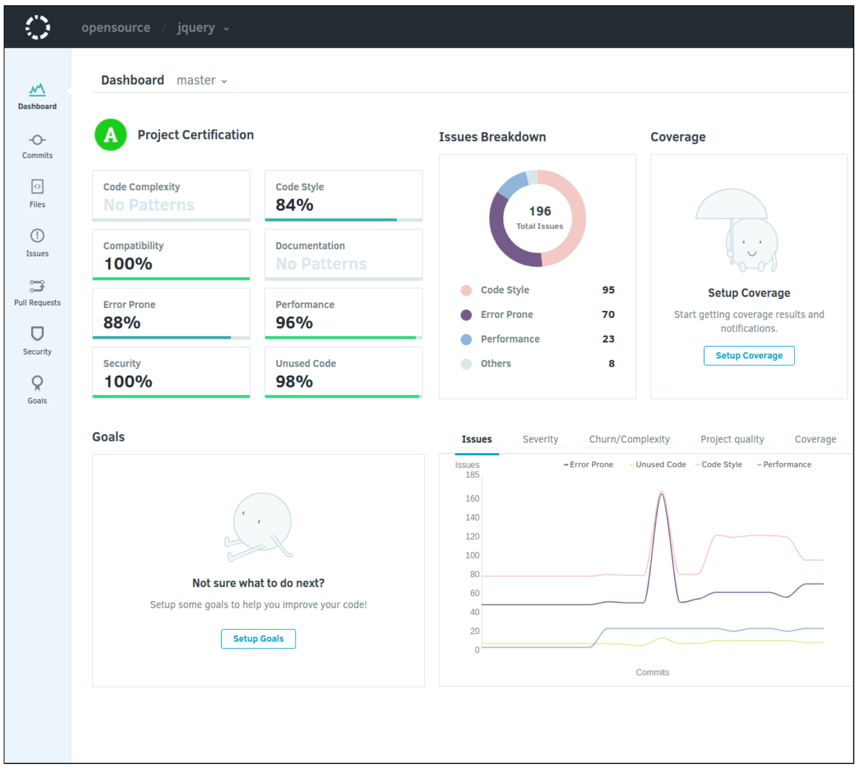
2.6.1 Code Review in Software Development

Because of its inherent closeness to the internet, and the existence of a large open source community, the field of software development is prone to having to use a lot of remote communication in the development process. This has resulted in a lot of tool support for remote communication. One example for these tools is Codacy, as seen in Figure 2.11[17].

Codacy is a web based tool for code review. Users add a project to the tool by connecting it with a Git repository from GitHub or any other Git server. Every time new code is pushed to the repository, Codacy then runs a series of automated tests for code quality, looking for anti-patterns in the code. The found anti-patterns are then represented graphically on the project's dashboard in the tool, ready for review. Users can also browse the source code to manually review parts of the project. If the project is sourced on GitHub, a direct connection to the repository can also be made. From any piece of code, as well as from the automatically found anti-patterns, a comment or issue on GitHub can easily be created.

There are many parallels between GlyphHub and Codacy. Both try to make adding a new or updated project into the platform as easy as possible, GlyphHub by automating compilation and installation, Codacy by giving a one-click integration for Git repositories. Both try to make reaching the right parts of a project to review them as easy as possible, GlyphHub by providing sample texts, Codacy by providing automated code analysis and a way to browse changes. And lastly, both try to give structure to feedback, GlyphHub by attaching all feedback to glyphs, Codacy by giving the chance to directly turn feedback into issues on GitHub.

[17]jquery on Codacy: `https://www.codacy.com/app/opensource/jquery/dashboard`, accessed on 2017-07-30.



Project Certification A

Code Complexity No Patterns
Code Style 84%
Compatibility 100%
Documentation No Patterns
Error Prone 88%
Performance 96%
Security 100%
Unused Code 98%

Issues Breakdown

196 Total Issues

Code Style	95
Error Prone	70
Performance	23
Others	8

Coverage

Setup Coverage
Start getting coverage results and notifications.

Goals

Not sure what to do next?
Setup some goals to help you improve your code!

Figure 2.11: The dashboard of the open-source project jQuery on Codacy. It shows automatically generated metrics about code quality such as performance and general code style, as well as statistics about the recent history of a project. With the navigation bar on the left, users can access more detailed information on the project.

2.6.2 Telepsychiatry

In the field of psychiatry, remote communication has started to play an important role over the last decade. Because of shortages of properly trained psychiatrists, especially in rural areas, many people living in these areas do not have access to face-to-face psychiatric treatment. Remote communication, in the form of telepsychiatry, has turned out to be an essential way to fight this disparity. For this, classical face-to-face therapy sessions are replaced with voice or video calls, oftentimes through dedicated software for telepsychiatry [6].

While there are no close parallels between telepsychiatry and GlyphHub, inspecting the field of telepsychiatry could still provide inspiration for ways of improving collaboration between type designers, not just for reviews, but also for other parts of the design process.

2.7 Future Work

To further improve type feedback, different ideas can be explored, some of them even within the context of GlyphHub:

2.7.0.1 Versioning Font Files and Comparing Glyph Versions

To allow reviewers to get an overview of a project's history of development, GlyphHub could allow a direct comparison between different versions of glyphs in said font. For this it would need a server sided version control system for fonts, as well as a client sided graphical representation of the differences between versions.

2.7.0.2 Integrating Feedback Into Source Files or Local Font Editors

An interesting feature for designers could be to allow them to directly merge proposed outline changes into their local source files. For this, the client would need to be able to manipulate local source files. Whether manipulating local source files is viable still needs to be evaluated, especially with proprietary source formats such as that of the Glyphs app.
An even more aggressive approach to integration of proposed changes is to directly integrate the feedback into the designers font editor of choice. This can be done by writing a plugin for the respective font editor, which would have to communicate with the GlyphHub server, display the server provided by GlyphHub, and merge proposed changes with the tool's font source files.

2.7.0.3 Tagging Feedback and User Ratings

Reviewers often want to reference other glyphs, projects, or comments. To make finding the right objects to reference easier, a tagging system in combination with a search function can be considered. To further sort the results of the search function, a way for users to rate feedback could be considered.

2.7.0.4 Building a Font Knowledge Base

Should GlyphHub find traction in the type design community and a tagging/rating system gets implemented, one could attempt to reuse the already structured and tagged knowledge about type design. It could be viable to build some form of wiki for type designers looking to learn about foreign writing systems. It could also be used to help find ways to automate glyph generation for complex writing systems, and through that help accelerate not just font review but the entire process of type design.

2.8 Conclusion

In this thesis, we have examined the process of reviewing type design projects. We have proposed ways to improve this process, and presented a prototypical implementation of these ideas in the form of the GlyphHub platform.
In theory, GlyphHub can bring a substantial improvement for the current review process in type design. It promises to improve the process by structuring given feedback, connecting reviewers with the designers' source files, and automating time consuming steps of the process. Whether GlyphHub can deliver on these promises remains to be seen. Further, more thorough testing of GlyphHub with a sufficiently large test group has to be carried out to be able to give a qualified statement on its effectiveness. Considering the circumstances of GlyphHub's development process, however, it can be expected to at least make progress on the right problems to improve font review.

3 Iterative Software Prototyping

Font design is a difficult process, even more so when designing fonts for multiple writing systems. Communication between font designers and language experts is crucial to designing correct and legible fonts. In this thesis, we describe how we used interviews and prototyping to develop a functional prototype for GlyphHub, a review tool for fonts. The most promising results from prototyping are presented and explained. We discuss which approaches worked well with designers and how the prototypes we used during the interviews might be designed better. Future developers of GlyphHub may use this paper as a basis to conduct further interviews more effectively.

3.1 Introduction

There are many different writing systems in the world used to write even more different languages. For all these languages written communication is essential. Most of this communication happens digitally. This means that digital fonts need to be available for more than just the Latin alphabet, but also for Greek, Arabic, Chinese and many other writing systems.

However, designing the necessary digital fonts is a difficult process. For each font all glyphs need to be designed by hand. This includes all letters and many other symbols, for example punctuation marks. In some writing systems, complex rules need to be defined so that the font is displayed correctly. The knowledge on how to correctly design a font is often only available from native speakers or language experts. Transferring this knowledge to the designers is a challenge, as the experts are often located in different countries.

In our project, we aimed to simplify this communication process in order to make the design of international fonts faster and easier. We talked to font designers who had experience in designing fonts for many different writing systems. We asked them about their experiences and showed them prototypes of our ideas on how to improve the font design process. With the insights, we gathered from the interviews we developed GlyphHub, a prototype for a font review platform. Designers can upload their font projects to the platform, so that other designers or experts can easily comment on its correctness and its aesthetics.

In this chapter we focus on the design and ideation process of GlyphHub. In section 3.2 we give an overview over important background knowledge that will be useful for understanding this chapter. Section 3.3 describes the interviews we had with designers and the prototypes we employed during them. We elaborate how the interviews helped us to explore the problem domain. Section 3.4 discusses whether the prototypes helped us to create a tool that meets the needs of font designers in terms of functionality and usability, by examining the value of the conducted interviews. An overview over related design processes is given in section 3.5. In section 3.6 we give a short overview of what the next steps in the design process could be.

3.2 Context: Usability Engineering

In this section we provide background knowledge on the methods we used during our ideation process. Two approaches to prototyping are presented, as well as different types of prototypes. Lastly, we introduce the test users we recruited for the interviews and user testing.

3.2.1 Approaches to Prototyping

We mainly used two different approaches to prototyping during our design process: Exploratory prototyping and evolutionary prototyping.

Exploratory prototyping, also referred to as throwaway or rapid prototyping, is used to "explore the requirements or certain details before or during development" [5]. Prototypes that are developed using this approach are not used in the final application and will be discarded. This allows designers to explore uncertain requirements without making the effort to fully integrate the feature into the system. The disadvantage of this approach is that all insights gained in this approach will still have to be implemented. We made use of this approach in the beginning, to quickly assess the needs of our users.

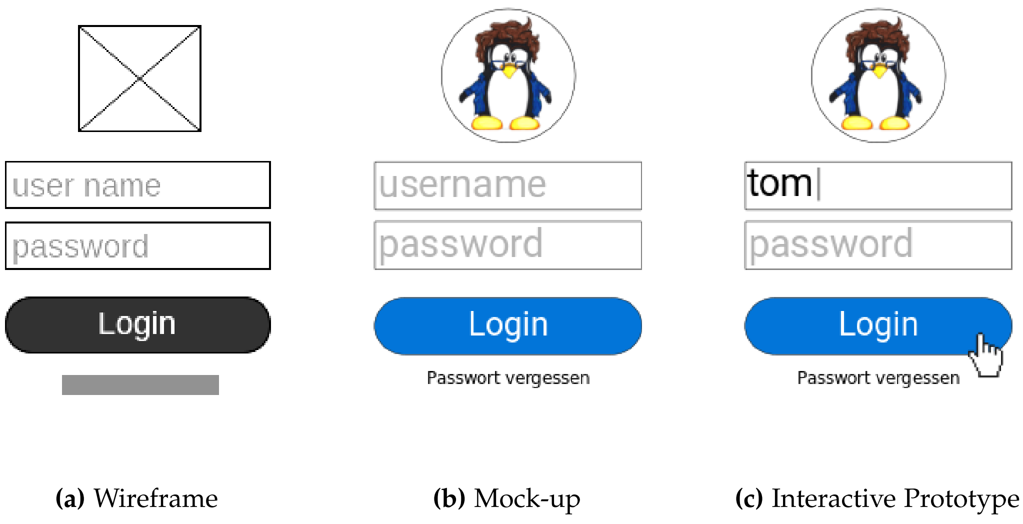
In evolutionary prototyping, most prototypes will be part of the final application. The benefits of this approach are that, while prototypes are more expensive to create, the final product does not need to be implemented, as it is the result of prototyping. Starting from those requirements that are well understood, a system is implemented and improved along the way. Once we had a better understanding of the user's needs, we started to use this approach.

3.2.2 Forms of prototypes

We differentiate between three different types of prototypes in this paper: Wireframes, mock-ups and interactive prototypes. A comparison of the three different forms can be seen in figure 3.1.

A wireframe is a schematic drawing of the user interface of an application. It depicts the page layout and the content structure. It does not display the final design of a product, and it does not have any functionality. It can be used to demonstrate what functions will be available later on and how interactions with the application are structured. The level of abstraction helps to stay focused on the structure and not concentrate on visual aspects of the design.

Mock-ups not only show the structure of an application, but also showcase the design. They give a realistic perspective as to what the finished product will look like, while still allowing revisions. Mock-ups can be designed on paper or on a



(a) Wireframe **(b)** Mock-up **(c)** Interactive Prototype

Figure 3.1: A login screen shown as a wireframe, a mock-up and an interactive prototype. The interactive part of the prototype is hinted at through the cursor and username being filled in.

computer. They are usually employed later in the design process, since they require more work.

An interactive prototype is an interactive model of the user experience. It can be clicked through by stakeholders and test users and lets them explore the user interface. Prototypes can consist of multiple wireframes or mock-ups. The interaction can be mocked by switching between these wireframes or mock-ups as needed. Interaction in a prototype can also be implemented in source code. This gives more possibilities, but also requires more work.

3.2.3 Test Users

Usability testing is the most reliable way to receive feedback on a user interface or application, as it provides direct input from the users. Before a user study can be conducted, however, test users need to be recruited. The test users should represent the intended user group of an application as good as possible [28, p175].

To recruit the right kind of test users it is often useful to develop a profile of the intended user of a system [10, pp.148]. We decided on the following criteria for our ideal test user:

- They should have experience in font design.
- They should have designed a font for at least one writing system other than Latin.
- They should be located in or close to Berlin.

With this persona in mind we started contacting people who fit these criteria.

We interviewed six different designers who fit the criteria but were still diverse in other aspects. Out of the six designers we were able to interview four were male and two female. They were born in five different countries, and had in total designed fonts for more than ten different writing systems. Experience in designing fonts for non-Latin writing systems was something we looked for in particular, since we wanted to improve this process with our application. While the designers we spoke to were open-minded about talking about our ideas and testing our prototypes, many stated that they wished to stay inside the environments they already knew. The designers would prefer not to include another tool into their workflow.

3.2.4 Prototyping Environment

The GlyphHub prototype was developed as an evolutionary prototype in Squeak/Smalltalk [17]. Squeak/Smalltalk is a live programming environment that uses the Morphic toolkit for its user interface [24]. Morphic is particularly useful for prototyping because of its live inspection features and easy composability. It formed a good base for our own user interface framework in which we extended Morphic with a custom set of user interface elements that were specifically suited for the font design domain. With these basic building blocks, we were able to quickly build and adapt our user interfaces. It was possible for us to directly edit the prototype and even live edit it during interviews. This proved useful, since sometimes errors prevented the user from testing a certain feature. With minimal effort, we were able to fix these problems during the interviews, so that we could receive helpful feedback on all features from the user.

3.3 Approach

Throughout this section we will present how we approached interviewing designers. There were two distinct rounds of interviews each with approximately the same designers.

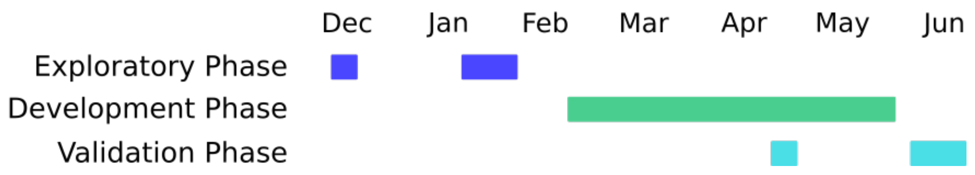


Figure 3.2: Timeline of the project. The project is divided into three phases. During the exploratory phase the first interviews took place, followed by the development phase of the GlpyhHub prototype and the validation phase in which the second round of interviews took place.

A timeline of the project can be seen in figure 3.2. The exploratory phase consisted of the first interviews we conducted. It was at the very beginning of the project. At this point we had very little knowledge of the font design domain. After the exploratory phase, we started work on the GlyphHub prototype in the development phase. The interviews during the validation phase took place after we had already worked on GlyphHub for several months, apart from one interview that was conducted during our development. We knew considerably more about fonts and the design process at this point.

This section is structured as follows: We examine what our goal during each interview phase was, describe what kind of artefacts we used and examine what we learned the interviews.

3.3.1 Exploring the Font Design Domain

The goal during the exploratory phase was to find a more concrete problem statement. Overall, we wanted to develop tools that would make designing international fonts easier. It should be an application that is useful to the designers, but could also be developed in a relatively short amount of time.

In preparation for the interviews we formulated questionnaires. In the first interviews, the questions were very general, for example: "What is your current process while designing a font?". The questions focused on the process of designing fonts, particularly on teamwork and designing fonts for multiple writing systems. We chose to focus on the design process because our knowledge of the domain as a whole was very limited and this process appeared to be the central component.

After several interviews, we were able to narrow our area of interest further down, because we had learned that communication was an essential part of the font design process. The questions became more specific, focusing on communication between font designers and people who give feedback on font projects. One question was, for example: "What form does the feedback you receive have? (Annotations, Pictures, Text, etc.)" We divided the questions into two groups: Questions primarily for font designers and questions for native speakers and experts that give feedback on font projects. The focus shifted towards the communication between designers, and other people involved in the font design process.

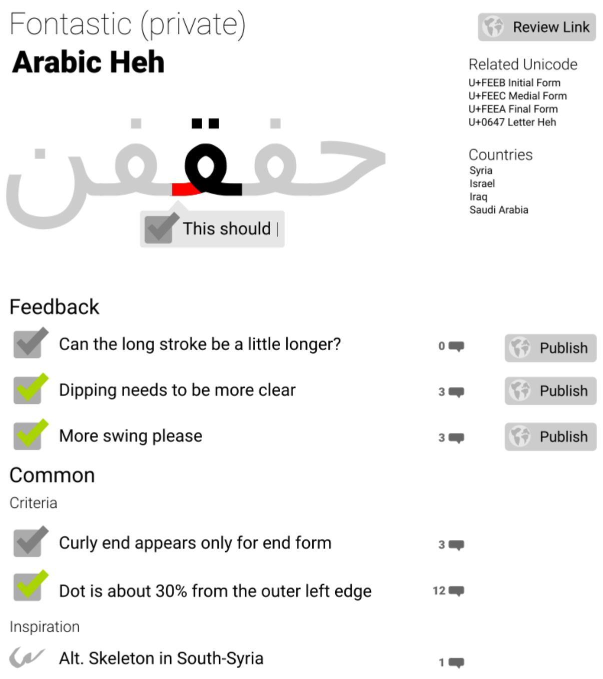
We prepared several mock-ups that were meant to make the communication between us and the designers easier and prevent misunderstandings. During the first few interviews two ideas that we wanted to explore further became apparent to us: (1) A font knowledge base and (2) a font review tool. We used mock-ups as a basis to explain our concept ideas to the font designers.

Figures 3.3 and 3.4 show the first approach, a place to collect knowledge about writing systems and font design, similar to a wiki. This sort of knowledge base would enable font designers, especially those who have little experience, to learn faster. They would no longer have to search for information on multiple websites and books. Instead all relevant knowledge would be collected in one place.

Figures 3.5 and 3.6 depict the second approach, a platform to upload fonts and comment on them. Designing international fonts requires feedback since most font designers are not familiar with all writing systems that they design fonts for. This review tool would therefore ease the communication between the font designers and experts who give feedback on font projects.

In the following we will examine these four mock-ups closely and describe the aspects that evoked the most interesting reactions, as well as problematic areas.

3.3.1.1 Mock-Ups for a Font Knowledge Base



Fontastic (private)

Arabic Heh

Review Link

Related Unicode
U+FEEB Initial Form
U+FEEC Medial Form
U+FEEA Final Form
U+0647 Letter Heh

Countries
Syria
Israel
Iraq
Saudi Arabia

This should |

Feedback

Can the long stroke be a little longer?	0	Publish
Dipping needs to be more clear	3	Publish
More swing please	3	Publish

Common

Criteria

Curly end appears only for end form	3
Dot is about 30% from the outer left edge	12

Inspiration

Alt. Skeleton in South-Syria	1

Figure 3.3: A mock-up for a *Font Knowledge Base* showing a glyph in a font uploaded by a designer. Several other glyphs of the font are shown as context. Additionally, feedback, common design criteria, and general information for the glyph are shown.

Figure 3.3 shows a mock-up for a font knowledge base in this case focusing on the glyph “Heh” of the Arabic writing system. Apart from the glyph itself it depicts several other glyphs to act as context and some general information about the glyph, for example related glyphs. In addition, it shows feedback that was received for this glyph, and general rules that have been collected by the community. Font designers are given the option to share this feedback by publishing it. The feedback will then appear in the common criteria, so that other designers can profit from it as well. We thought this would be a useful feature since many times, especially when designing fonts for unfamiliar writing systems, feedback is given on incorrect design, rather than aesthetics. This general advice is applicable to all font projects.

The context given for the glyph was an explicit requirement we had gathered in previous interviews and font designers were pleased with the implementation of it. During the interviews, we found out that the general information about the glyph was not needed by the designers as they did not find it relevant. In many cases it wasn't clear to the font designers what the difference between common criteria and direct feedback was.

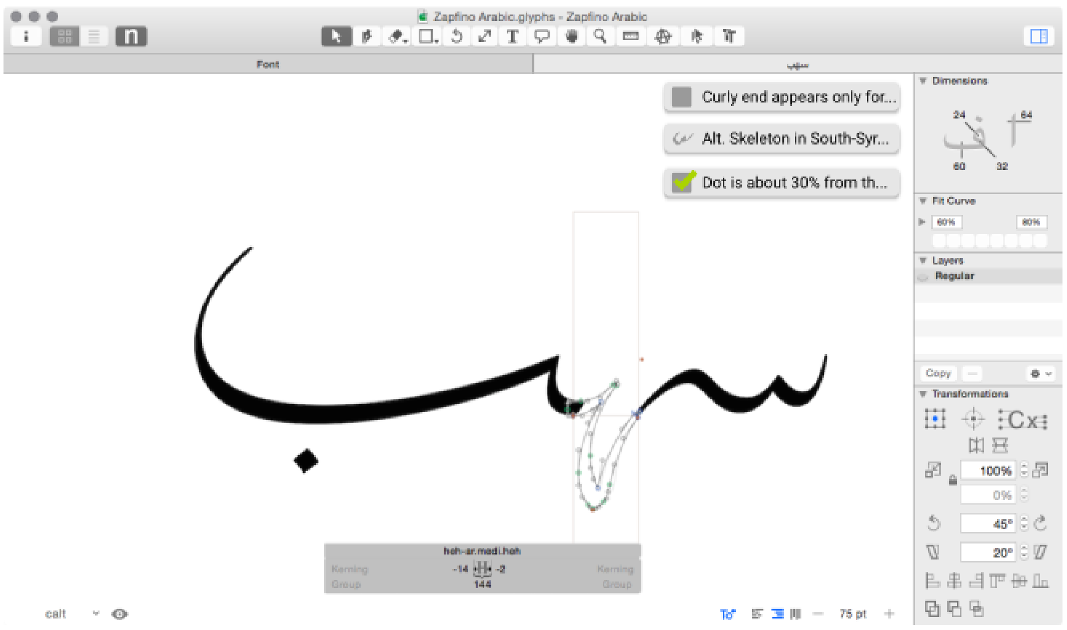
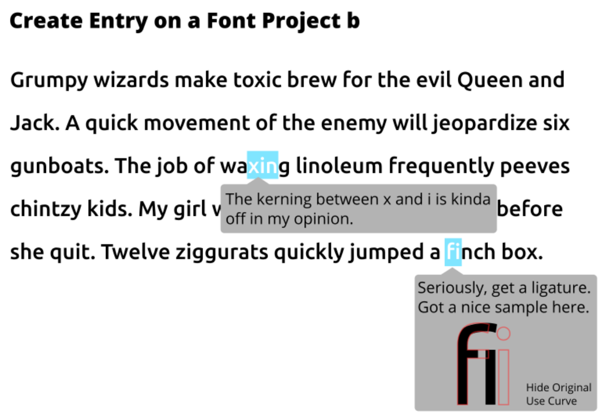

Curly end appears only for...
Alt. Skeleton in South-Syr...
Dot is about 30% from th...

Figure 3.4: A mock-up enhancing Glyphs, a tool for font creation, with comments that help during the design process. The comments are shown in the upper right corner next to the toolbar. In the middle, the glyph that is currently being edited is shown.

Figure 3.4 shows how such a knowledge base might integrate itself into Glyphs[1], a tool for font creation. The comments are shown inside the application alongside the glyph editor. This approach has the benefit of designers being able to stay in a well-known environment, something they had explicitly wished for. The idea of integrating such a social component into their daily work was well received by the designers. Some even suggested to create a feature to share a screenshot of current work on the platform.


Create Entry on a Font Project b

Grumpy wizards make toxic brew for the evil Queen and Jack. A quick movement of the enemy will jeopardize six gunboats. The job of waxing linoleum frequently peeves chintzy kids. My girl v before she quit. Twelve ziggurats quickly jumped a finch box.

The kerning between x and i is kinda off in my opinion.

Seriously, get a ligature. Got a nice sample here.

Figure 3.5: A mock-up showing comments on a sample text. Text passages that are commented are highlighted in blue. The comments are depicted in gray dropdowns and consist of text or changed outlines.

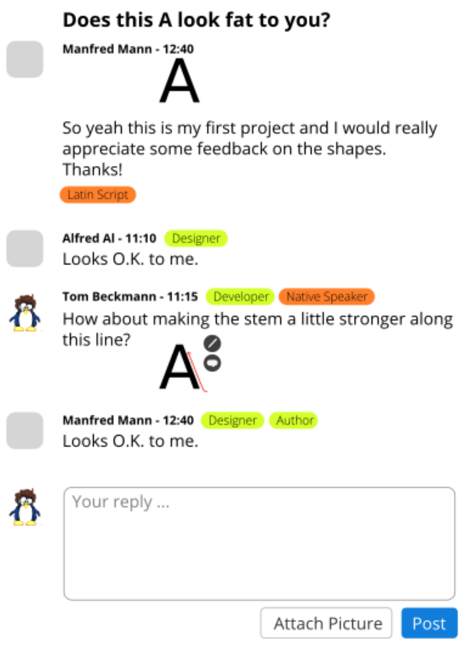
3.3.1.2 Mock-Ups for a Font Review Platform

In figure 3.5 an idea for what commenting on a sample text could look like is shown. Passages of text or single glyphs can be selected and commented on. In addition to writing text, the outlines of glyphs can also be edited and the changes can be proposed to the owner of the font. Font designers generally liked the idea, but they did not understand the subtleties of it. It was not clear that the labels *Use Curve* and *Hide Original* next to the new outlines were meant as buttons.

Comments in the Font review tool could be displayed in threads as shown in Figure 3.6. Three people have posted in this comment thread: the original author of the font and two reviewers. Everyone has labels concerning their roles, for example thread author, designer or native speaker. These labels were meant to make it easier to put the feedback received into a bigger context. One designer we talked to, however, had concerns that these labels could unjustly influence the perception of the feedback. For example, only the feedback from native speakers might be considered even though other designers might have the same expertise.

Once we completed this phase of our project we had sufficient knowledge of the font design process to identify one of its main problems. We knew that designers struggled with getting feedback on their font projects, especially when designing international fonts. We decided to focus purely on reviewing from this point on. While designers were interested in both approaches we had shown to them, we realized that we did not have the necessary expert knowledge to create a font knowledge base. We could provide the basic framework, but the most important part, the expert knowledge to fill this knowledge base, would have to be supplied by others. A review tool for fonts could still be extended by also providing general information.

[1]Glyphs, `https://glyphsapp.com/`, visited on 2017-07-26.



Does this A look fat to you?

Manfred Mann - 12:40

So yeah this is my first project and I would really appreciate some feedback on the shapes. Thanks!

Latin Script

Alfred Al - 11:10 Designer

Looks O.K. to me.

Tom Beckmann - 11:15 Developer Native Speaker

How about making the stem a little stronger along this line?

Manfred Mann - 12:40 Designer Author

Looks O.K. to me.

Figure 3.6: A mock-up of a comment thread in a font review tool discussing the glyph **A**. A user suggested to change the glyph and attached a picture to his comment.

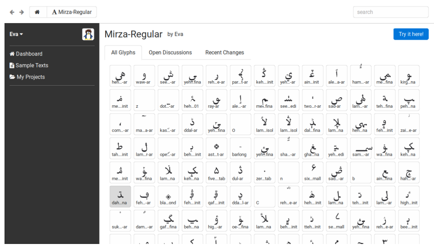
In the next phase of our project, the development phase, we developed the GlyphHub prototype. GlyphHub is a dedicated review platform for fonts that allows designers to upload font projects. Experts and native speakers from all over the world can then give feedback by commenting on single glyphs or text passages. The feedback is presented in a structured form to designers so they can easily see the problems and edit the font.

3.3.2 Evolutionary Prototyping on a Font Review Tool

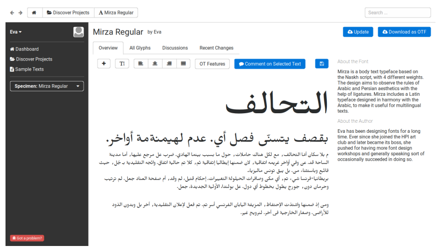
The goal during the validation phase was to validate whether the conclusions we had drawn from the exploratory phase were correct. Will the prototype, that we built upon these assumptions, make designing international fonts easier? Will designers understand how to use the tool? In addition, we also wanted to get some feedback on features we were planning to add in the future and re-evaluate how commenting on sample texts could work.

In the first interview, we gave a short demonstration of the GlyphHub prototype. One important aspect we learned was that sample texts are very important when giving feedback on a font. However, they were not yet featured in our application. Before having any more interviews, we restructured the application and introduced sample texts.

As depicted in Figure 3.7 we changed the font overview after the interview. Instead of first seeing the overview of all the glyphs in the font, as shown in



(a) The original font overview page. All glyphs of the font are shown.



(b) The new font overview page. A sample text is prominently featured on the page.

Figure 3.7: The original and the new version of the font overview page side by side

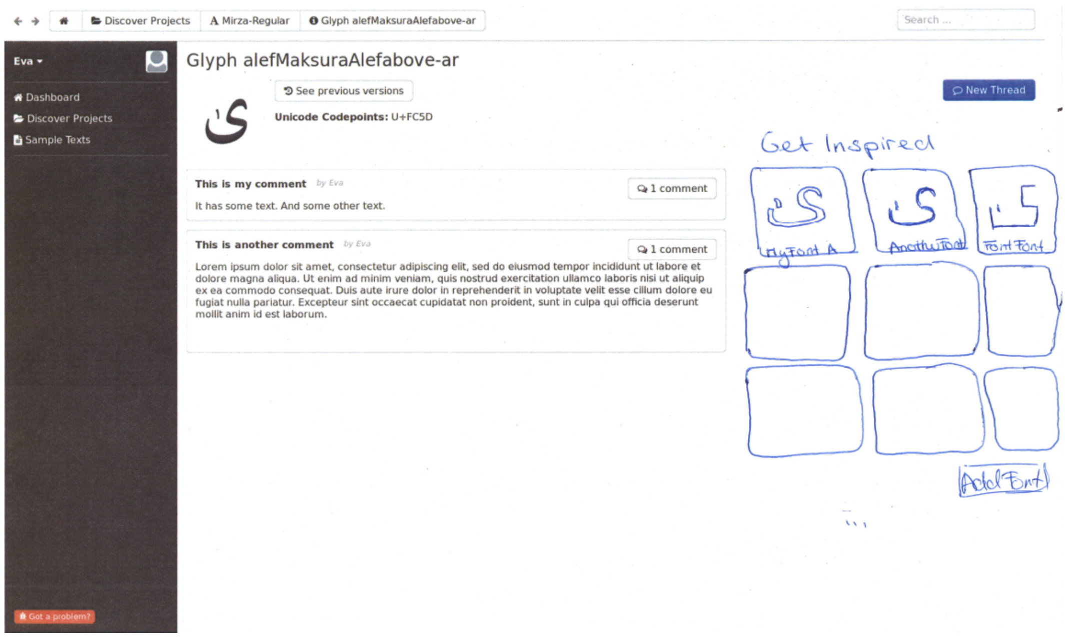
Figure 3.7a, reviewers are now introduced to the project with a sample text chosen by the designer. Additionally, some information about the font and the designer is given. All glyphs can still be seen by selecting the "Overview" tab. This can be seen in Figure 3.7b.

In the following interviews we chose to interview the same designers again, since they were already familiar with the basic premise of our project. We tested the usability by giving the designers small challenges, for example "Upload a new font project", that they had to solve during our interviews. All the tasks were small enough to be accomplished during our relatively short interviews while still being meaningful enough to give us valuable feedback on our application. We watched their interaction with the application closely during these tasks and noted in which places they experienced difficulties. We asked our test users to *think aloud*. They described what they were doing and thinking while solving our tasks. This method lends itself well to collect high quality data from even a small amount of test users [28, p.195]. The interviewers should not attempt to take over when the user does not immediately know how to solve a task but only observe and listen [35, p.191].

We learned that many of our buttons were not as clear as we had thought. For example, the button to add a paragraph to a text was frequently mistaken as a button to increase the text size. On multiple occasions, designers suggested we add a feature that was already implemented but was not found by the designers. This was especially problematic since there was no help or documentation available inside the application.

Font designer often expected GlyphHub to support the same interactions as their font editor. This includes, for example, jumping to certain glyphs in the overview as keyboard shortcuts. The fact that these interactions were not supported in GlyphHub seemed to confuse the designers.

Opening a new page could take a longer time due to loading operations being executed and blocking the page. These loading operations, however, were not indicated to the user, so that font designers often were irritated by the fact that application did not react. This prevented us from getting more feedback on the functionality of GlyphHub as the designers focused on these technical problems, that were not part of the idea of the prototype.



This is my comment *by Eva*

It has some text. And some other text.

This is another comment *by Eva*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 3.8: A mock-up added to GlyphHub to inspire users with their design. The glyphs is shown on the top left above its comments. On the right side several other glyphs that can inspire the user are drawn in pen.

We also prepared some mock-ups for this phase, as seen in Figure 3.8. We wanted to find out if a certain feature would make a good addition to our tool without having to implement it first. The proposed feature was meant to inspire font designers in their design process by showing them, for any glyph they wish, versions from other fonts. This feature would also help reviewers by letting them compare a glyph to others and give more explicit feedback.

The prototype was well understood, but designers tended to focus on parts that were not part of the feature we wanted to discuss. While the things we wanted to talk about stood out, as they were drawn in pen, it did not draw enough attention to them. Feedback on the feature revealed that inspiration would not be required after already designing the glyph.

We also experimented with some other types of prototypes, for example wireframes. We tried to get the designers to purely focus on the feature at hand rather than improvements to the user interface. Figure 3.9 shows such a wireframe. We wanted to demonstrate a proposed change to the sidebar, which can be seen on the left side. All other elements were held abstract and did not have text. This led to a confusion for the designers. They did not understand what the page was meant to do, and even though it was not important they focused on this part, trying to understand it.

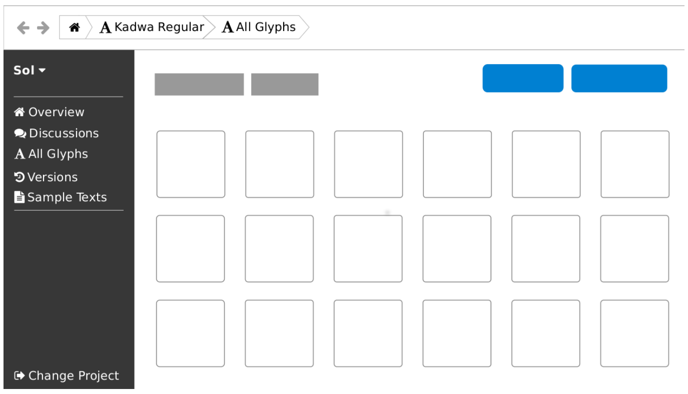


Figure 3.9: A wireframe of a page in GlyphHub, supporting focus on the sidebar on the left side. All other elements are kept on a more abstract level.

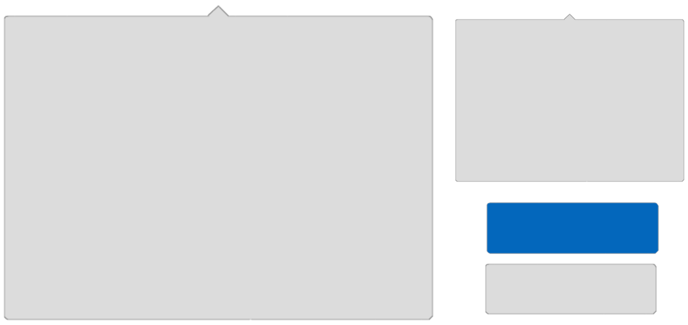


Figure 3.10: A model kit consisting of two dropdown in different sizes, a primary and a secondary button. This kit can be used for paper prototyping.

Lastly, we employed a model kit approach during one of our interviews. We printed out several parts that make up a user interface, for example buttons, dropdowns, empty GlyphHub pages, and sample texts. The buttons and dropdowns we printed can be seen in Figure 3.10. With these tools, we asked font designers to imagine how commenting on a text would look like ideally.

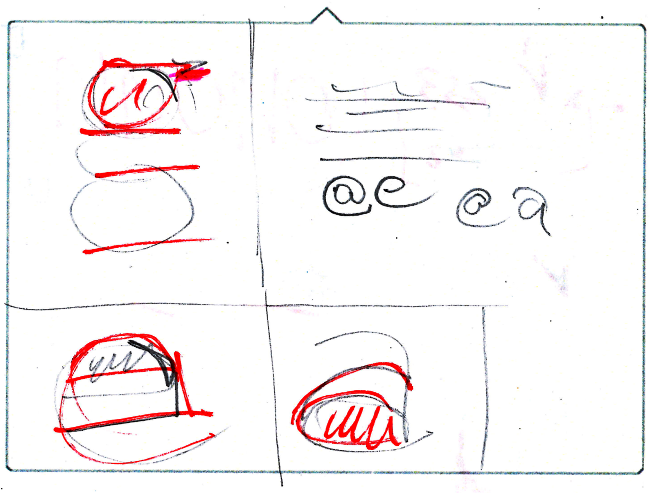


Figure 3.11: A sketch of a dropdown used for commenting. Inside the dropdown the glyph g is shown in the top left corner. Additional glyphs, e and a, were added by typing @e or @a. Some text is also shown on the top.

In some ways, the result was quite similar to what we had imagined in the beginning, a dropdown that appears when selecting text. However, the contents of this dropdown as drawn by the designer differed substantially from our mock-ups in the beginning. As seen in Figure 3.11 the dropdown contains the glyph being commented on, in this example the glyph g and some text. Additional glyphs, e and a, can be added by typing @e or @a. The designers can draw onto the glyphs to better visualize their feedback.

We realized that many parts of the model kit we had printed were not necessary. Buttons were not used at all, they were rather quickly sketched directly onto the paper. The dropdowns proved useful as they inspired creativity. We quickly ran out of prepared dropdowns and instead used pieces of paper cut to approximately the same size. In general, the prepared pieces were not really necessary but, nonetheless, they helped to make the intent of this exercise clearer.

Overall, we received valuable feedback. The interviews gave us a good overview of what features were already usable and where we could still improve. We learned that many features, for example sketching on a glyph as part of a comment, were not easy enough to discover. In addition, GlyphHub as a whole lacked in usability. We discovered a new way to implement commenting. We learned that inspiration was not needed after having designed a glyph but might be useful during commenting.

Based on all our interviews we were able to gain a solid knowledge of the font design domain and build a prototype to solve one of the key problems of the domain.

3.4 Discussion

In this section we evaluate whether the interviews helped us to create a useful prototype for GlyphHub. We discuss which prototypes received useful feedback and which were less helpful. We also examine the costs of the interviews, from our perspective and from the designers' perspective.

The interviews helped us to get a good understanding for the user's needs. This provided a good foundation for our project in a short amount of time. We learned about the font design process. Without talking to designers we probably would not have been able to develop anything at all, since we had no knowledge of the domain. In some aspects, however, we could have used them more effectively.

3.4.1 Value of Interviews

In both interview phases, we were able to reach the goal we had set for ourselves. In the exploratory phase, we were able to define a concrete problem statement: the review process for font design needs to be improved, so that more international fonts can be created. However, it stands to debate if we could have had a more relevant problem statement. As said in section 3.3, after only very few interviews we narrowed our questionnaire down to two topics. This was mostly due to the first interviewee being very focused, as they had recently made some bad experiences with receiving feedback on their font. When prompted almost all designers could tell us of some similar experiences they had had with reviewing. But would they have chosen to focus on it if we had not prompted them to? Maybe they would have come up with different areas in which they would have seen the need for improvement.

On the other hand, if we had never narrowed down our questions, we would not have been able to find a specific focus area at all. By asking these questions we learned more about the specific requirements for a font review tool.

In the interviews during the validation we tried to validate our decisions. Generally, font designers were pleased with the GlyphHub prototype. They did say it would make their process easier, but only after being explicitly asked. These questions might have introduced a bias. Avoiding questions that can create a bias is an important skill that any interviewer should have [10, pp.250]. However, we did not practice this before our interviews.

Another problem with these interviews was that most of the feedback could not be incorporated into the GlyphHub prototype anymore. This is mostly due to time constraints. Other aspects, for example the new way of commenting, might be very difficult to implement, as we would have to change many existing procedures. As such, it would have been helpful to have these interviews earlier in the implementation phase. Ideally these studies would have been conducted before we implemented commenting for the first time.

3.4.2 Value of Prototypes

Prototypes were an important tool during our interviews. They acted as a communication device among us and the font designers and made it easier to share our ideas. However, the design could have been clearer in many cases.

Our prototypes, especially in the beginning, showed too many details. This led to the designers being overwhelmed and aspects that were important to us were often not discussed. Removing unnecessary details might have helped. Instead of showing a detailed mock-up, designing a wireframe that displayed the structure and the content might have been enough. However, especially in the first interviews, we did not know what the important aspects were. Often the aspects that seemed less important to us raised discussions we had not expected. For example the tags on people in Figure 3.6 led to an interesting discussion about the need for moderation in comment threads.

For mock-ups that had buttons, it would have been useful to show what these buttons would do once clicked. For example in Figure 3.5, it was not clear that *Use Curve* and *Hide Original* were buttons that changed the view of the "fi" glyphs. Providing mock-ups that demonstrate what would happen upon clicking these buttons would have made it easier to explain. We learned that keeping things visual instead of having to resort to long explanations made the communication easier. This is especially true for an inherently visual domain, as font design is.

We never quite found the right balance between wireframes and mock-ups. Mock-ups lead to the focus shifting to unimportant subtleties of the design. Whereas wireframes led to the designers being confused as to what they were seeing. This might be a problem caused by the fact that we only showed wireframes after we had already shown some mock-ups. The different level of detail might have been confusing.

One of the most interesting uses of prototypes was the model kit we used in the validation phase. It gave us a valuable insight into what a designer needs when making a comment. This insight came only from one designer however. Using the model kit approach with other designers could have revealed that they need different content when giving feedback. For this reason, it would have been interesting to conduct more interviews using the model kit. These interviews could also have taken place much earlier in the process, ideally in the exploratory phase of interviews. Having this input at this point would have enabled us to implement this way of commenting and receiving feedback from other designers.

3.4.3 Cost of Interviews and Prototypes

The cost of interviews can be determined by the time they cost us and the designers, as no other resources were involved for either party.

The preparation time for the interviews was relatively short, especially for the output we generated with them. We conducted them before and after our implementation phase, so that we had adequate time to prepare them. Most of the prototypes we used during the interviews were created in only a few hours. With this little effort, we were able to prevent days of development on an ultimately useless idea.

For the font designers we interviewed this differs slightly. They did not directly gain anything from the interviews. For them they were costly, as they prevented them from working on their font projects. This was one of the reasons we did not conduct more interviews during our development phase.

3.5 Related Work

In this section we briefly describe two approaches to exploring the user's needs and designing user-centered applications: Design Thinking and User Centered Design.

3.5.1 Design Thinking

A popular method for innovation is Design Thinking. The method relies on three things: the place, the process and the people [15, p.17].

The Design Thinking process consists of three to seven steps, depending on the literature. The number of steps does not influence the underlying principles behind the method. Only the amount of detail, which is given to the steps, changes slightly [30, p.113]. The six steps of Design Thinking are, according to the Stanford School of Design Thinking[2]: Understand, Observe, Define Point of View, Ideate, Prototype and Test [37]. It is a non-linear process, so the order of the steps may be changed and steps can be repeated throughout the process.

During the Understanding phase research and interviews with experts are conducted. In the next step, Observe, people and their interactions are observed. People are asked as to what they are doing and why. Defining a Point of View helps to focus on people's needs. Often the phrase "How can we ..." is used to make suggestions on how to impact people's lives. Ideation encourages design thinkers to come up with as many ideas as possible for the problem they defined as their point of view. Ideas can be far-fetched and unrealistic; quantity is most important. After having explored all possible directions, ideas can be grouped and elaborated. During Prototyping and Testing different prototypes are rapidly developed and

[2]Stanford School of Design Thinking, `https://dschool.stanford.edu/`, visited on 2017-07-30.

tested. These last two steps are repeated so that the Design Thinking team can iterate on the feedback they received during testing.

Design Thinking is more than a process, it is a work culture as well. Design Thinking Spaces are especially designed for collaborative work. Teams should always be interdisciplinary, as each person brings their know-how and working methods to the team. Individual design thinkers should be versatile as well. The ideal design thinker is "T-shaped", meaning someone who has deep knowledge of their domain, but also has a wide array of other interests [40].

Design Thinking works by finding many different approaches to a solution first before narrowing it down to only a few. In comparison, we only developed two approaches and went straight on to the implementation, without first searching for more possible solutions. We also did not test our prototypes as often as the Design Thinking process suggests. Only after a development phase, that lasted for the majority of our project, we presented the GlyphHub prototype to test users. If we had done this earlier, it would have been easier to adapt to the feedback we received during the interviews, especially since the feedback became apparent very quickly.

3.5.2 User Centered Design

User Centered Design is a term used to describe "design processes in which end-users influence how a design takes shape" [1]. It can involve a variety of different methods, but the resulting products should always be based on the needs of the user. They should be usable and understandable [29, p.188].

Nielsen defines usability as five components: Learnability, Efficiency, Memorability, Errors and Satisfaction [28, p.26]. The user interface should be easy to learn and easy to remember. It should be efficient and satisfying to use. Additionally, it should not be prone to errors and it should be possible to recover from all errors.

Designing usable interfaces is often done by exploring user requirements. User requirements describe the needed features and actions from the user perspectives. These requirements are gathered by talking to users. In later stages of the process the recognized user requirements are then validated by doing user testing.

User centered design involves the users from the beginning to the end of the development [32]. Input from the user should be the basis for all design decisions.

The process used to develop GlyphHub was partly user centered. We analysed user requirements and conducted usability studies, but only at the start and end of the project. In between we did not have contact with our users and made many decisions about functionality without the user's input.

3.6 Future Work

The GlyphHub prototype was well received by the font designers we interviewed. In many areas it might still need improvement. Some areas of improvement were already discovered through usability testing as described in subsection 3.3.2. Due to the nature of our project we were not able to do any large-scale user testing. This might uncover more potential areas where GlyphHub could still be improved. If we had the opportunity to conduct more user studies, we would focus on two aspects: real world data and user interaction.

3.6.0.1 Real World Data

During our development phase and interviews we always operated on sample data. While we uploaded actual font projects, all of them were already published and did not need further feedback. The comment threads and sample texts were all created by us and could only aim to imitate the actually used data. We can assume that real data would differ quite a bit from our samples. To prevent unwanted behaviour conducting some user studies would be beneficial. Giving font designers the opportunity to upload a real project and invite reviewers to comment on it, would provide us with the needed data.

Communities to give feedback on fonts and advice on the designing process have already been formed. One of the most notable places where such a community meets is the Google Fonts Discussions group[3]. A large number of possible testers could be found here.

During these studies, the frequency of each application error can be logged. This provides a good overview of where problems happen and what might have caused them thus giving the development team a chance to fix them. In addition, logging the user's interaction with the application will provide an opportunity for statistical analysis. These logs could include which features are used frequently and which are only used rarely. The benefit of using this approach is that it requires no active participation of the user and is unobtrusive [31, p.378]. The user's privacy should always be respected during these analyses. All logged data should be anonymous and actions should not be able to be traced back to a specific user.

Features that are only rarely used should be investigated as to the discoverability and necessity of such a feature [28, p.217]. To investigate this, follow-up interviews could be conducted. Users would be given the opportunity to state why they did not use a certain feature and give further feedback. This limits the chances of misinterpreting the collected data.

3.6.0.2 User Interaction

One key aspect of our design was the social component. However, during our user tests, we did not have an opportunity to try out how designers react to the social component since we only interviewed one designer at a time. We tried to imitate this behaviour by uploading their font projects and commenting on them ourselves, but we could not imitate real conversations this way. Studies with multiple users would be very interesting.

A setup for this kind of test might simply consist of two font designers sitting in different rooms at the same time both using the tool. However, it might not

[3]Google Fonts Discussion Group,
`https://groups.google.com/forum/#!forum/googlefonts-discuss`, visited on 2017-07-26.

be possible to generate a lot of feedback like that, since reviewing is a process that takes its time and cannot always be condensed into thirty minutes or an hour. This is especially true since the font designer whose projects are being reviewed would need adequate time to respond to the review, which might involve correcting outlines in their fonts.

Instead, testing the application during active use would provide us with the needed data. User tests in this manner would have to be conducted over a longer period of time. Ideally the projects would be reviewed on our platform during the entire process – from the first sketches to the finished font.

While working with GlyphHub, font designer and reviewer could fill in questionnaires. Questionnaires have the advantage of requiring less effort than interviews. Collocation and finding a fitting time is not necessary, the questionnaires can be filled in by the test users and simply be send back via mail. This makes it possible to get data from a large number of test users.

Due to this indirect nature of questionnaires it is recommended to use rating scales, for example a Likert-type scale as seen in Figure 3.12. Open questions, for instance "What did you think of this feature?", inspire the user to answer with full sentences, rather than single words. However, the answer might be even harder to understand [28, pp.212]. Rating scales are less ambiguous since the possible answers are predefined.

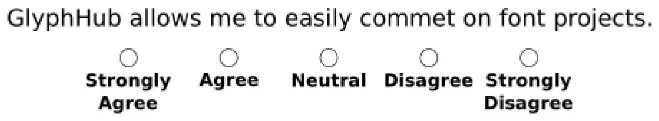


Figure 3.12: A possible question about GlyphHub for further user testing as a Likert-type Scale.

A possible question for such a questionnaire is shown in figure 3.12. Participants only have to choose the option that represents their opinion. These questionnaires can help designers identify problem areas. As the user only needs minimal effort to give feedback questionnaires can be admitted multiple times throughout the process. The answers can then be compared and used to demonstrate improvements or deterioration to the user interface [35, p.169].

3.7 Conclusion

In this chapter, we described the design and ideation process of GlyphHub. GlyphHub is a review tool for fonts, that aims to simplify the font design process by making communication between designers and experts faster and easier. We elaborated how we gathered the necessary domain knowledge through interviews with font designers. The methods we used during the interviews, as well as the prototypes we designed for them, were explained and their effectiveness was discussed. We learned that it is important to focus on the essential part of the user interface and not include too many details in a prototype.

Further interviews were conducted after developing the GlyphHub prototype. During these interviews, one approach proved particularly useful: We asked a font designer to show us how commenting on a sample text should look like. To help them we provided a small model kit, consisting of basic user interface elements, so that they only had to be arranged by the designers. This approach could be employed in interviews with other designers as well. We also provided an overview of what other techniques proved useful during interviews. In addition, areas and techniques for further user testing have been proposed. Future GlyphHub developers may use this knowledge to improve the prototype into a fully functional font review tool.

A Interview Questionaires

The questionnaires were used in the exploratory interview phase, as described in subsection 3.3.1.

A.1 Original Questionnaires

- What is your current process while designing a font?
  - Do you often make drafts with pen and paper or digitally?
  - Where do you start?
- Which tools do you use?
  - How many different ones?
  - Which tool do you use for which step?
  - Do they interact (properly)?
- Do you always work alone on fonts?
  - Would you like to work in a group?
  - Do you see any problems with working in a group?
  - What would those problems be?
  - What benefits do you see in working as a group?
- Is there anything that bothers you about your current process?
- Do you remember when you started designing fonts?
  - Was it easy to start?

    - Would a tutorial be helpful in your opinion?
    - What should be included in such a tutorial?

- What type of fonts do you design? (Latin, Greek, Chinese...)
    - Are there differences in the process for different scripts?
- What do you think of our process idea?
- Do you write down which languages you want to support with a font?
- Would you like some kind of feedback on how close you are to being done?
- Do reviewers (especially font experts) only look at large test-documents with example texts (or similar) or also at single glyphs/ligatures etc.?

A.2 Updated Questionnaire

This questionnaire was originally written in German. A translated version is avaible below in subsubsection A.2.2.

A.2.1 German Version

Fragen für Designer

- Beziehen Sie Feedback zu Ihren Font-Projekten?
  Von wem (Experten/Kunden)?
- Über welche Art und Weise findet das Feedback statt?
  (Test-PDFs/Mail/Github) Gedruckt/nicht gedruckt?
- Zu welchen Aspekten der Font bekommen Sie hilfreiches/unnötiges Feedback?
  (Glyph-Formen/tatsächliche Fehler/Ästhetik/Kerning/OT-Features)
- Wann beziehen Sie Feedback (sobald die ersten Skizzen/Glyphen/Sätze da sind, täglich/wöchentlich?)
- In welcher Form findet das Feedback statt?
  (Annotationen in den Glyphen/Glyph-Diffs/Textuell/Zeichnungen)
- Für welche Skripte haben Sie bisher Fonts entwickelt?
- Legen Sie fest, welche Sprachen eine Font unterstützen soll? Wie finden Sie heraus, wie viele dieser Sprachen diese Font schon unterstützt?

Für eine nicht-Muttersprache:

- Woher haben Sie das nötige Fachwissen bezogen, um eine korrekte Font für diese Sprache zu entwickeln
  (Literatur/„einfach“ die Sprache gelernt/Experten)?

- Ist es eine Herausforderung die OpenType-Features korrekt zu beschreiben? Mit welchen Features beginnen Sie? Welche sind Ihnen nicht so wichtig?
- Kann es passieren, dass einzelne Glyphen vom Schriftsystem her „falsch“ beschrieben sind? Fernab von Lesbarkeit oder Ästhetik. Wie häufig recherchieren Sie die Regeln eines Systems (nach)?
- Würde es Ihnen helfen, wenn ein Reviewer Ihnen während des DesignProzesses „über die Schulter gucken“ könnte und Hilfe gibt? Könnten Sie sich vorstellen, solche Hilfe statt persönlich direkt im Werkzeug eingebettet zu bekommen?
  Würden Sie solche Hilfe gern stündlich/täglich/wöchentlich einholen? Oder kontinuierlich?
- Vergleichen Sie manchmal die aktuelle Version Ihrer Font mit einer älteren Version?
  - Ja: Wie (mit welchem Tool)? Wobei hilft das? Wie groß ist der „Altersunterschied“ zwischen den verglichenen Versionen? Ist es immer ein Vergleich über die ganze Font/einen Satz/einen Glyph?
  - Nein: Warum nicht? Kennen Sie Werkzeuge, die das tun würden? Sind Ihnen diese zu kompliziert? Was ist daran zu kompliziert? Hätten Sie einen Verbesserungsvorschlag?
- Angenommen ein Reviewer könnte direkt die Kurven eines Glyphs ändern. Meinen Sie, dass hätte eine Qualität, die Sie direkt in Ihr Projekt übernehmen würden?
  - Ja: Haben Sie schon einmal aktiv mit einem Designer in einem Font-Projekt zusammengearbeitet?
  - Nein: Warum nicht? Wäre es dennoch hilfreich die Unterschiede zwischen Ihrem Glyph und dem veränderten zu sehen?
- Stellen Sie sich eine Sammlung von Wissen über Glyphen bzw. Kombinationen von Glyphen vor. Dort könnte Inspiration (gute/schlechte Beispiele), Kriterien zur Korrektheit des Glyphen und Trivia über den Glyph selbst gesammelt werden.
  - Haben Sie so etwas ähnliches schon irgendwo gesehen? Ihre persönlichen Notizen vielleicht?

Fragen für Reviewer

- Wem geben Sie Feedback? Einzelnen Designern/Gruppen?
- Über welche Art und Weise findet das Feedback statt?
  (Test-PDFs/Mail/Github)? Gedruckt/nicht gedruckt
- Zu welchen Aspekten der Font geben Sie Feedback?
  (Glyph-Formen/tatsächliche Fehler/Ästhetik/Kerning/OT-Features)

- Entstehen daraus zum Teil Gespräche oder wird das Feedback i.d.R. einfach direkt umgesetzt?
- Wann geben Sie Feedback?
  (bereits zu den ersten Skizzen/Sätzen, täglich/wöchentlich)
- In welcher Form findet das Feedback statt?
  (Annotationen in den Glyphen/Glyph-Diffs/Textuell/Zeichnungen)
- Gibt es Fehler, die praktisch bei jedem Projekt wieder auftreten? Gibt es Feedback, dass sozusagen „verallgemeinerbar“ ist?
- Welche Tools benutzen Sie für Ihre Reviews? (InDesign/diffing/...)
- Wie prüfen Sie, ob das Feedback korrekt umgesetzt wurde? (PDF zurück mit den Änderungen/alles nochmal neu durchgehen)

A.2.2 English Version
Questions for Designers

- Do you receive feedback on your font design projects? If so who is giving you feedback (Experts/Customers)?
- In what manner do you receive feedback (Sample-PDFs/Mail/Github, Printed/Digital)?
- On which aspects do you receive helpful/unnecessary feedback?
  (Glyph-Forms/Incorrect design/Aesthetics/Kerning/OT-Features)?
- When and how often do you receive feedback (as soon as the first sketches/glyphs/sentences are available, daily/weekly)?
- What form does the feedback you receive have? (Annotations, Pictures, Text, etc.)?
- For which writing systems have you designed fonts?
- Do you decide up front which languages a font should support? How do you know how many languages a font already supports?

Concerning a language that you do not speak as a native language:

- Where did you gather the necessary expert knowledge to design a correct font for this language? (Literature/"simply" study the language/Experts)?
- Is it a challenge to correctly define OpenType features? What features do you start with? Which ones are less important?
- Is it possible that single glyphs are specified incorrectly? Apart from legibility or aesthetics, how often do you research the rules of a writing system?

- Would it help you if a reviewer would "look over your shoulder" and give you advice during your design process?
- Could you imagine receiving feedback embedded directly into the font creation tool of your choice? How often would you like to receive feedback? Hourly/daily/weekly? Or continuously?
- Do you sometimes compare the current version of the font with older versions?
  - If so: How (using which tools)? How does this help? How large is the time difference between the different versions? Do you always compare the complete font/a sentence/a glyph?
  - If not: Why not? Do you know tools that support this? Are these tools too complicated for you? What is complicated about them? Do you have any suggestions for improvement?
- Assuming a reviewer could edit the curves of your glyphs directly. Do you think they would be of a high enough quality so that you would use them in your project?
  - If so: Have you ever collaborated with a font designer on a font project?
  - If not: Why not? Would it be helpful to see the differences between your glyph and the changed glyph?
- Imagine a collection of knowledge about glyphs or combinations of glyphs. You could find inspiration (good/bad examples), criteria for the correctness of glyphs and interesting trivia about each glyph here.
  - Have you seen something like this before? In your personal notes for example?

Questions for Reviewers

- Who do you give feedback to? Individual designers/groups of designers?
- In what manner do you give feedback?
  (Test-PDFs/Mail/Github) Printed/digital?
- Which aspects do you give feedback on?
  (Glyph-Forms/incorrect design/aesthetics/kerning/OT-Features)?
- Does feedback invoke longer conversations or do you generally just implement the feedback?
- When and how often do you give feedback? (on the first sketches/sentences, daily/weekly)?
- In what form do you give feedback? (annotations in Glyphs/Glyph-Diffs/in text form/sketches)?

- Are there mistakes that happen in all projects? Is there feedback that is universally applicable?
- Which tools do you use for reviews? (InDesign/diffing/...)
- How do you check if the feedback you gave was implemented correctly? (PDF with changes/look through all glyphs again)

4 Viability of Complex Font Rendering in Live Environments

To be accessible to international users, computers have to support several scripts. To display them we need complex text rendering. Live environments have a unique kind of interactivity and debugging possibilities. We tested several approaches to integrate complex text rendering into one specific live environment. Based on that we propose an alteration to the default text rendering of the chosen environment and discuss the applicability of the findings to others.

4.1 Introduction

Font Rendering can be found on all computers. Whether it is a website, a text editor, or a programming environment, text has to be rendered dynamically. Originally, text consisted only of a small set of ASCII characters. Until today, that character set has been expanded to Unicode, which includes characters from languages from all over the world. Those numerous new characters brought new challenges for font rendering as more complex mechanisms have to be supported. A word in Arabic or Devanagari does not only have letters, words are combined based on the characters into composites. The same character may need a different localisation in English or Polish. A font for programming may automatically replace groups of characters with their logical equivalent for visual purposes. Today text rendering has to support all this and more.

Text rendering originally started with *simple text rendering*. To support basic Latin, for example the characters of the ASCII table, relatively few glyphs are needed. Once a digital version of a glyph has been drawn by a designer, it can be duplicated as much as needed. A picture looks best in the resolution it was created in, but having one text size or a few is enough for basic text interaction. The visual information for a set of characters with a shared style is saved in a *font file*. Like block letters the pictures defined in the font can simply be placed beside each other to create text. This is a fast and easy way to render text on computers that support some languages, for example English. This process becomes more complicated if more requirements are placed on it. If the text should have optimal quality at each size, it is not enough to have static pictures for each text element. If more scripts should be supported, more glyphs have to be designed and more complicated glyph placement rules have to be integrated. International scripts work in different ways, for example Devanagari or Arabic can be compared to Latin written in cursive. The glyphs cannot simply be placed beside each other, all characters in a word influence each other. New font file formats emerged that can save different kinds of visual information and features that influence the text. To render these fonts correctly, *complex text rendering* is needed.

As part of our bachelor project we created a font review tool, GlyphHub, in a *live environment*. Font designers make fonts for all kinds of languages and use cases. These fonts which are created for complex rendering also have to be reviewed with text created with complex text rendering. For reviewers and designers it is important to see how the font behaves in sample texts and on different systems. Creating such a tool in an object-oriented live environment allows changes to all objects in it without delay. All objects can be observed and interacted with live. Advantages of this are better discoverability and enhanced debugging.

The next section will contain more context on the problem of complex rendering. After that follows a description of chosen criteria for complex rendering, then different approaches for rendering text in a live environment. Following that, the approaches are compared based on the criteria. The next section contains a proposed alteration to the used live environment. In the subsequent section, the comparison and how these concepts apply to other live environments will be discussed. The last sections contain related work and a conclusion with an outlook.

4.2 Context

This section provides further motivation and background on complex text rendering. First there will be a short introduction to text rendering in the live environment *Squeak*[1] [17]. All implementations in this paper are based on Squeak version 5.1. Part of the code was created as part of our bachelor project[2]. Then complex text rendering will be explained in more detail. The difference between simple and complex rendering will be shown utilizing an example. All factors that are important for complex text rendering will be explained in more detail.

4.2.1 Font Rendering in Squeak

Squeak is a live environment. It is an object-oriented programming system and a dialect of Smalltalk [14]. All objects in Squeak can be inspected and interacted with live, there is no separate compiling process between changes to the code and interacting with the system. In its current rendering process, several objects are involved that represent the involved entities in text rendering.

The paragraph object is essential for text rendering in Squeak. It manages the text attributes set by the user and the text layout to be ready for display. Widgets that display text, for example the `TextMorph` from the morphic framework, have a paragraph object. The `TextMorph` contains a canvas, a picture, of the rendered text.

[1]Squeak environment: `http://squeak.org`, visited on 2017-07-27.
[2]GlyphHub: `https://github.com/HPI-SWA-Lab/GlyphHub`, visited on 2017-07-27.

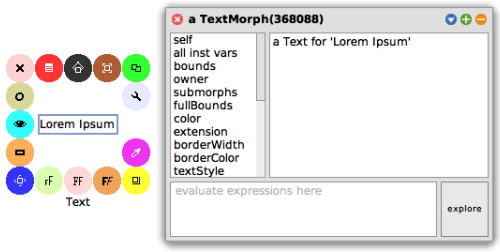


Figure 4.1: The picture shows a `TextMorph` displaying the sample text 'Lorem Ipsum' in Squeak. On the left side is the `TextMorph` itself, surrounded by a halo. Objects in Squeak can be selected to open a halo menu which has buttons for certain interactions, for example deleting the object, and shows the type of the object, in this case 'text'. On the right side is an inspect window for the `TextMorph`. It shows all attributes of the object, the attribute names being in the left upper field and the value of the currently selected attribute in the right upper field. Below that is a field for code input and a button to open an explore window for the object.

Text elements, like glyphs, are not morph objects. The class usually used for this is `NewParagraph`.

The paragraph does not compute the layout by itself. A composer object uses information of the text such as the user input and font to derive all character positions. One example for a composer is the `TextComposer`.

A font contains information needed for digital text in one style. It stores the visual information as *glyphs* and related information such as *typographic features*. The fonts in Squeak are represented by a font object. This object provides an interface with which the information is provided to others, for example to the composer object. All fonts are subclasses of `AbstractFont` which defines the interface.

4.2.2 Complex Font Rendering

To display text on digital device, text rendering is needed. A font file contains all visual information needed for this. Designers can define the look of their glyphs according to which style they want their font to have.

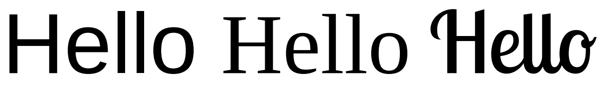


Figure 4.2: The picture shows the example text 'hello' rendered in the fonts sans-serif, serif and lobster. The 'H' of the word on the left is very simple and without ornaments. The word in the middle has serifs and a more noticeable different thickness for parts of the 'H' compared to the first one. The last 'H' features an decorative element on the left side to fit the style of the design of the font.

There are different ways of rendering fonts. For example, we want to render a text with one word written in Latin and one in Arabic. We could use simple rendering which like block letters places glyphs beside each other. For first text rendering of the ASCII characters this was enough. This way of rendering text is very fast in its execution and easy to implement because of its simplicity.

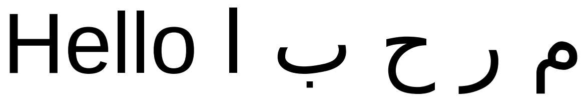


Figure 4.3: The picture shows the result when simple text rendering is used for the word 'hello' in English and Arabic. The English version is correct; the Arabic version lacks glyph composition. In the Arabic script, separate glyphs might still be readable as letters but only composite glyphs can form comprehensible words.

But as seen in Figure 4.3, what works for English does not necessarily work for Arabic. Arabic writing is in certain regards similiar to Latin written in cursive. The glyphs connect to each other and change their shape when written together. The glyphs by themselves are still correct but as a word they are incomprehensible. Complex text rendering is needed to display the text correctly.

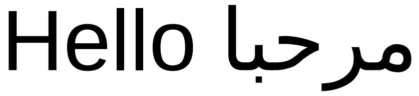


Figure 4.4: The picture shows the result when simple text rendering is used for the word 'hello' in English and Arabic. Both words are correctly displayed. Compared to the version with simple text rendering, small alterations can be seen which happen when the glyphs are combined instead of single.

There are several factors that influence how the given text should look like rendered. These are the given font, text size, typographic features, script and language as well as the text itself and its context[3].

The font is the basis for the rendering, it contains information for all its supported glyphs. Without it, even simple rendering would not work. There are different ways to include the visual information of the glyphs in a font. A vector font stores the information as bezier curves which are drawn when a text with the font is rendered.

[3]Google Talk on Complex Text Rendering Video: `https://www.youtube.com/watch?v=Plhnr7mPhxw`, visited on 2017-07-27,
Google Talk on Complex Text Rendering Slides:
`https://webengineshackfest.org/2015/slides/recent-blink-improvements-in-text-and-layout-by-dominik-rottsches.pdf`, visited on 2017-07-27.

A bitmap font on the other hand contains prerendered pictures of all glyphs which are placed according to the input.

Text can be rendered in different sizes, for example a headline is usually bigger than plain text. Vector fonts can be scaled without problem, but bitmap fonts may loose quality if they do not have the particular size prerendered. That is because the scaling of rasterized pictures may create artifacts. In addition, rendering approaches working with vector fonts are usually optimized for rendering different sizes in a readable way even on different kinds of displaying hardware.

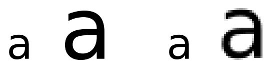


Figure 4.5: The picture shows 'a' rendered four times in different sizes. The two glyphs on the left are rendered directly at their respective size. The smaller glyph from the pair on the right side was also rendered directly, but the bigger version was created by manually scaling up the smaller glyph. Because of this the glyph on the right has artifacts which make it look blurry.

Text is not always written from left to right. Other writing systems also need other directions. This influences spacing between glyphs, how the context is processed and more. Complex rendering should support left to right, right to left, top to bottom and bottom to top.

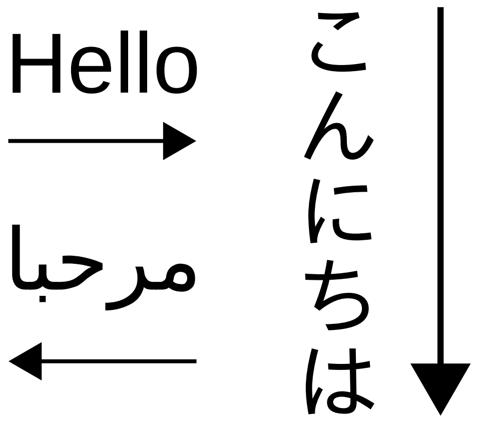


Figure 4.6: The picture shows 'hello' in English, Arabic, and Japanese. The English word is written from left to right, the Arabic one from right to left and Japanese is written from top to bottom.

A font has typographic features, which contain information on how certain input should be rendered. While users can activate or deactivate such features for design reasons, some features are required to make the rendered text comprehensible. For example, Arabic glyphs can only form meaningful words when *character composition* is active. On the other hand, the *smallcaps* feature is just a design element in headlines, which does not affect comprehensibility. In fonts, typographic features are described in a certain format such as the commonly known *OpenType*[4] format. Such formats specify the set of available features. Ligatures and other substitution features, for example, replace a part of an input text that is in a specific order with another glyph. A feature such as kerning does not change which glyphs are rendered, but where they are placed in relation to each other.

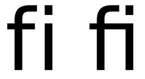


Figure 4.7: The picture shows 'fi' two times. On the left side it is rendered without typographic features, two glyphs are placed next to each other. The version on the right side has the ligature feature activated. This replaces the two separate glyphs with another glyph with a special design which in this case visually connects the 'f' and the 'i'.

The script and language of the text is also important. Different scripts are often in different font files, so using fallback fonts has to work correctly. Some scripts are only used by one language, some are used by multiple. Scripts for multiple languages may have special localisations of glyphs. Some languages need multiple scripts to be fully supported, Japanese for example needs Kanji, Hiragana, Katakana, and parts of Latin including numbers[5].

Lastly, we need the text and its context to render correctly. Even if only part of a text is rendered, it is influenced by its context since features should still be in affect even if part of the result is not visibly rendered. Note that text can be split into words at specific delimiters such as white-space characters without loosing the required context.

The chosen font and the rest of the user input provide all neccessary information for this, but only a *shaper* object can create the result. The shaper is responsible for determining which glyphs should be drawn at which position. To incorporate all influences, the shaper has to at least work on a word basis and not on a character basis.

The shaper provides correct width information for single characters and character compositions. Without the shaper, text might be drawn virtually correct, but can be cut off or unwanted white space can appear.

[4]List of OpenType Features: `https://www.microsoft.com/typography/otspec/featurelist.htm`, visited on 2017-07-27.

[5]Japanese Writing System: `http://www.japaneseprofessor.com/lessons/beginning/intro-japanese-writing-system/`, visited on 2017-07-27.

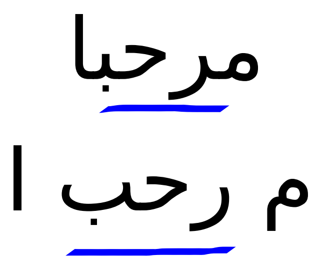


Figure 4.8: The picture shows 'hello' in Arabic rendered two times. The text that is wanted is underlined, the rest is context that would normally not be rendered. The upper word is rendered correctly because it takes the context into consideration, even if the context may not be shown. The lower version ignores the context which makes it look like a separate, different word. That is not desired, since a part of a word should still be rendered correctly if the rest of it is not shown.

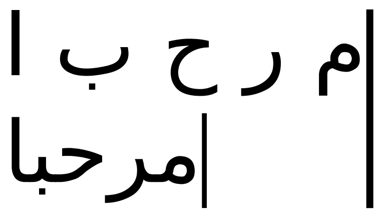


Figure 4.9: The picture shows 'hello' in Arabic with simple and complex rendering followed by lines. There is space between the two lines which has an influence on the layout. If the layout uses the width of the upper version and renders the lower one, it would lead to unwanted white space.

4.3 Criteria of Font Rendering

This section will elaborate on three criteria for text rendering based on the context. The first subsection deals with correct complex text rendering and what that means for this paper. The focus is to have correct rendering for international fonts. After that, the focus is on quality and different techniques used in text rendering. The last subsection deals with not only having objects for the whole rendered text but also for the rendered text elements.

4.3.1 Correct Complex Rendering

There are different aspects of correctness of rendered text, for example the result should always fit on the canvas it is displayed on, no text should be cut-off.

The most basic aspect of being correct is whether all given glyphs can be rendered. Without this, there can be no legible text. For this paper it is still correct if only specific font formats are supported by a rendering approach.

Better results are a result of supporting typographic features. Some scripts can only be rendered correctly from a linguistic view with these features. If the font contains wrong information or is missing some, the result may be wrong for a reader, but it is technically correct from the rendering perspective. Any implementation should interpret *all* information in the font because skipping of typographic features can yield wrong results.

It is also important to able to render glyphs not covered by the chosen font in another fallback font. A font usually only covers a part of all possible characters, for example enough to cover the Japanese language. This can be seen in the Google Noto project which consists of many fonts in the same style that aspire to cover all of the unicode characters[6]. Unsupported characters should not be rendered as undefined glyphs if possible. An undefined glyph would be one special fallback character, for example a square, that is used for all glyphs the font does not support. A font as similar as possible to the given one should be chosen to render them. To display all text in a legible way, it would be enough to choose any font that covers the missing glyphs.

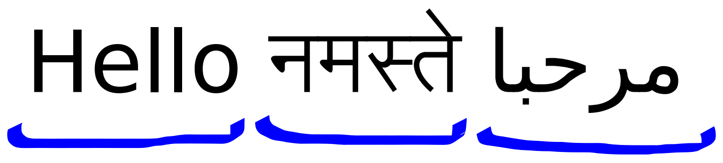


Figure 4.10: The picture shows 'hello' in English, Nepalese, and Arabic. The words are underlined to show where delimeters are in the example. Whether the words are rendered after each other or all at the same time does not have an impact on the result, because context is important on a word basis.

[6]Google Noto: `https://www.google.com/get/noto/`, visited on 2017-07-27.

For this chapter, a completely correct rendered text adheres to all given information on a word basis. No information should be ignored which means that simple text rendering is incorrect by default as it ignores typographic features. The rendered text should fit onto the given canvas. If possible, input not supported by the current font should be rendered with a fallback font.

4.3.2 Quality

Rendering results can have different levels of quality. This is mainly affected by which techniques are used to render the text.

The simplest way of rasterizing is drawing completely black lines. The great contrast creates something called an *aliased* picture. These pictures tend to be less legible especially at small sizes.

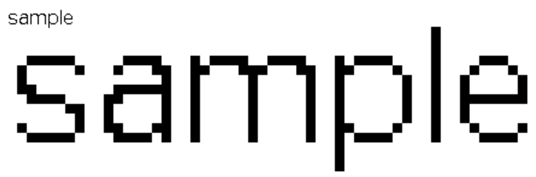


Figure 4.11: The picture shows 'sample' rendered very basic without antialiasing. The text was rendered at a smaller size and then scaled up to show details. All pixels are completely black or white which leads to sharp edges. (Source: `https://upload.wikimedia.org/wikipedia/commons/8/8c/Rasterization-simple.png`, visited on 2017-07-27)

To increase legibility, *anti-aliasing* can be used. This technique creates grey-scale pictures. The hue of each pixel depends on how much of the glyph is on the pixel based on the glyph's vectors. This can lead to blurry results if the glyphs are placed in unfortunate ways.

With *hinting* the blurriness can be reduced. These hints have to be in the information of the font file. After utilizing the hints, the result has clearer lines. This technique is compatible with simple rasterization as well as anti-aliasing.

A pixel on a screen is actually divided into differently colored subpixel. A technique called *subpixel rendering* uses this to let curves appear smoother which increases resolution. This can be combined with anti-aliasing and hinting. This technique is dependent on the display hardware of the user. Usually there are three subpixels placed horizontally in red, green, and blue color. However, this is not universally defined and may differ between screens. The arrangement may differ, for example a vertical placement is also possible. Number and colour of the subpixels can also vary. Subpixel rendering can only be used successfully if placement and colour of all subpixels is known and supported by the process. If this is ignored, it may create a more blurry result.

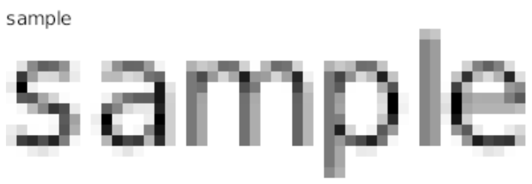


Figure 4.12: The picture shows 'sample' rendered with antialiasing. The text was rendered at a smaller size and then scaled up to show details. The result is grey-scale, the hue depending on how the curves of the glyph pass through the pixel. At normal size the text is a bit blurry because of the many light tones. (Source: `https://upload.wikimedia.org/wikipedia/commons/0/09/Rasterization-antialiasing-without-hinting-2.png`, visited on 2017-07-27)

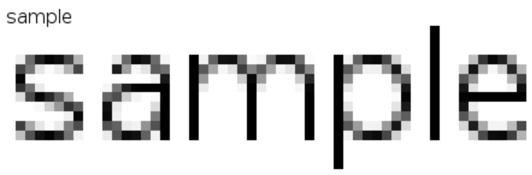


Figure 4.13: The picture shows 'sample' rendered with antialiasing and hinting. The text was rendered at a smaller size and then scaled up to show details. The hinting helps determining which pixels should be darker to create a less blurry result. (Source: `https://upload.wikimedia.org/wikipedia/en/e/ec/Rasterization-antialiasing.png`, visited on 2017-07-27)

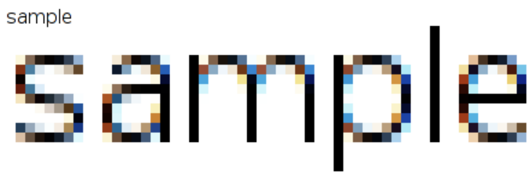


Figure 4.14: The picture shows 'sample' rendered with subpixel rendering. The text was rendered at a smaller size and then scaled up to show details. A pixel contains three subpixels in different colours. Using these can create a clearer image. This technique is dependent on the displaying hardware. (Source: `https://upload.wikimedia.org/wikipedia/commons/0/0b/Rasterization-subpixel-RGB.png`, visited on 2017-07-27)

The type of font also affects quality. Vector fonts contain the lines and curves of the glyphs. This makes them scalable without pixellation. But rasterizing the curves dynamically requires more processing than placing pictures which usually leads to longer rendering time. A bitmap font on the other hand stores already rendered or drawn bitmaps for each glyph. Because of that they are generally very fast to render. But as mentioned in the context, they are not arbitrarily scalable, they look best at their original size. Bitmap fonts also cannot adjust dynamically to different displaying hardware.

For this chapter supporting quality means supporting all techniques at any text size.

4.3.3 Interactivity

Standard font rendering puts the entire rendered text on one picture. The user can select and edit the text or parts of it, but there is only one underlying object.

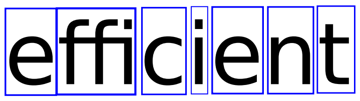


Figure 4.15: The picture shows the text ‘efficient’. All separate glyphs are encased by rectangles to show which glyph objects would be needed to create the text. Noticeable ‘ffi’ is one glyph because it is a ligature.

As an alternative it is possible to render certain elements as their own objects. One way would be to have all involved glyphs as their own objects. This allows for better interactivity and explorability. It also possibly creates new debugging possibilities, both for the code of the text rendering process as well as the font itself and its features.

4.4 Different Rendering Approaches

There are several ways to render text in the Squeak environment. In the following section, three different implementations are looked at in more detail. One that is part of Squeaks default text rendering, using the `StrikeFont` and the `BitBlt` plugin. `StrikeFont` is a bitmap font which supports very simple text rendering. Another approach is using an already implemented external library for the text rendering. A subsection deals with such a library: *Pango*[7] [38]. The last way to render text is using the *Balloon* renderer, made in general to render curves and lines, to render vector fonts.

[7]Pango: `http://www.pango.org/`, visited on 2017-07-27.

There are several font file formats. One is *UFO*, the unified font object[8]. One special trait of .ufo files is that the curves and lines of the glyphs are not in binary but in their original form, making them easy to read. Compiled, binary font files are for example the .otf files of the *OpenType* format[9].

The live environment Squeak runs in a virtual machine. This virtual machine can be extendend with plugins[10]. External plugins allow to run code from outside the virtual machine, for example a function written in C. Externally run code cannot be debugged like code in the Squeak environment can and usually requires the virtual machine to pause while it is processed. Because of that external plugins are usually used for primitives that cannot be implemented directly in the virtual machine or functionality that is significantly faster when run outside the live environment. All rendering approaches use plugins which can be seen in Figure 4.16, but how much of the text rendering and layouting is done externally or in Squeak varies.

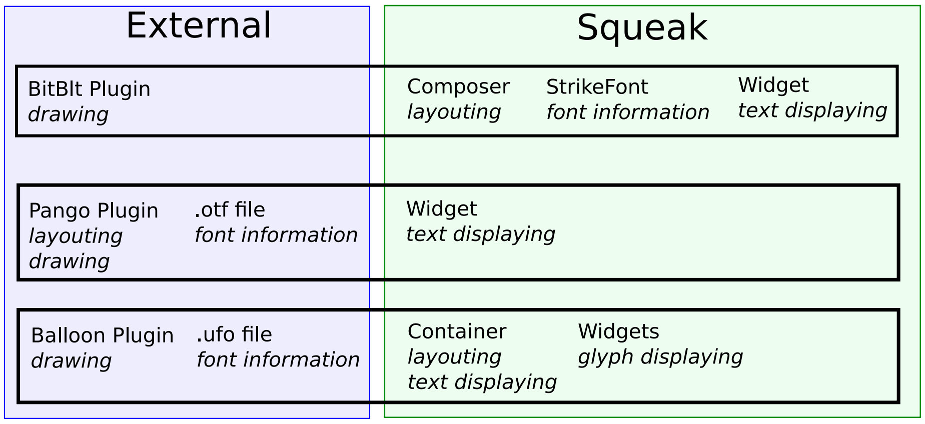


Figure 4.16: The picture shows which parts of the rendering process happen externally and which in Squeak. The colored block on the left side is external functionality and on the right side is Squeak. All parts of one approach are placed on a horizontal line and surrounded by a frame. The involved entities have their names written and their purpose below the name in italic. Layouting in this case means both fitting the text into the given space as well as choosing the correct glyphs to render based on typographic features and other input. The `StrikeFont` tries to keep as much as possible of the rendering process in Squeak and using Balloon also only has drawing and the font file itself outside the environment. Pango on the other hand has most of the logic externally.

[8]UFO: `http://unifiedfontobject.org/versions/ufo3/`, visited on 2017-07-27.
[9]OpenType: `http://www.microsoft.com/en-us/Typography/OpenTypeSpecification.aspx`, visited on 2017-07-27.
[10]Squeak VM Plugins: `http://wiki.squeak.org/squeak/1448`, visited on 2017-07-27.

4.4.1 StrikeFont and BitBltPlugin

A `StrikeFont` is a font object in Squeak and can be used to render text. As a subclass of `AbstractFont` it is part of the current Squeak environment and compatible with the current rendering mechanisms. It is a bitmap font so it does not have its glyphs as vectors but as a prerendered bitmaps. Glyphs from the bitmap are cut out and placed beside each other to render text by the `BitBlt` plugin. They are no external font files, the data is in objects in Squeak.

4.4.2 Pango Plugin

Pango is text layouting and rendering library written in C. It is possible to use it as an external plugin for rendering but it is currently not integrated into the default rendering mechanisms of Squeak. Together with *Cairo*[11] Pango can render text directly but it is also possible to only use Pango to process the text input and draw the result with another module. In this case, Pango would act as a shaper, using the input parameters to determine the necessary glyphs and their positions. Through a plugin interface pango is given the text that should be rendered, the font it should use, the size of the canvas it can render on and other textstyle information, for example if certain OpenType features should be used such as smallcaps. It then returns the rendered text which is displayed by a widget in Squeak. The used font files have to be compatible with Pango. For example, this can be the .otf format which makes Pango compatible with vector fonts.

4.4.3 Balloon Plugin Renderer

Vector fonts can also be rendered with the Balloon renderer. The Balloon package in the Squeak environment supports rendering of bezier curves, but the text rendering based on this is not compatible with the current Squeak rendering mechanisms. This renderer currently only works with the .ufo format, because in this format all curve information of the glyphs is accessible. It can only render specific glyphs which are then given a layout by a container widget. The glyphs themselves are their own objects. This layout at the moment only forces a new line when the next glyph object would no longer fit into the container.

4.5 Comparison

In this section the rendering approaches will be compared based on the chosen criteria correct complex text rendering, high quality support and interactivity through objects. In the first subsection the correctness of complex text rendering is the focus which only one approach fully supports. The next section deals with quality. All

[11]Cairo: `https://www.cairographics.org/`, visited on 2017-07-27.

approaches have different levels of quality, Pango being the most versatile and `StrikeFont` being dependent on the prerendering of the glyphs. The last subsection looks at text elements as objects for enhanced interactivity. This is currently only part of the approach using the Balloon renderer. A summarized result can be seen in a summary Table 4.1.

Table 4.1: The table shows which approach supports which criteria completely. The focus of the bitmap font StrikeFont is to have a fast and basic text rendering, because of that it supports none of the criteria. Pango was used because it supports complex text rendering and compatibility with high quality rendering. The Balloon renderer was not created with text rendering in mind, but was used here to create text with interactivity through objects.

Approach	Correct Complex Rendering	Quality	Interactivity
StrikeFont and BitBltPlugin	no	no	no
Pango Plugin	yes	yes	no
Balloon Plugin Renderer	no	no	yes

4.5.1 Correct Complex Rendering

The approaches fulfill different levels of correct text rendering. All support simple rendering, they can draw their glyphs on a canvas without something being cut-off.

Of the rendering approaches only Pango supports complex rendering completely. It draws a correct result on the canvas based on the input. Glyphs unsupported by the given font are replaced with fallback data if fonts containing them are available.

The other two approaches only allow for simple rendering without taking typographic features into consideration. The Balloon text-rendering implementation can draw all vectors of a glyph of a given font, but it currently does not have the logic to correctly translate input information containing OpenType features into a correct output drawing. Note that – even if not implemented at the time of writing – another object could preprocess the input and then provide glyphs with correct coordinates to Balloon.

`StrikeFont` is a bitmap font that only supports simple rendering with additional kerning which makes it incorrect. Squeak text rendering does not allow for correct complex rendering because of a wrong distribution of responsibilities.

4.5.2 Quality

Quality varies between the approaches and the circumstances they are used under, such as the quality of the prerendering of the bitmap font or the used display hardware.

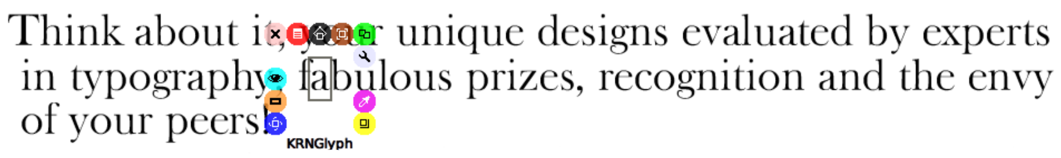

Think about it, your unique designs evaluated by experts in typography, fabulous prizes, recognition and the envy of your peers!

Figure 4.17: The picture shows an example text from a kerning tool prototype which uses the Balloon renderer. All glyphs are separate objects, one of these glyph has a halo to show this.

Pango supports very high quality. Because the drawing component can be exchanged, it supports all techniques of drawing glyphs. It also supports vector fonts, allowing optimal quality at all sizes. Since it is an external plugin, it may be slower in general, the complex rendering also needs more processing than a simple approach.

Since the Balloon Renderer draws vector fonts, scaling is no problem for quality, but it lacks the text rendering specific logic of the Pango approach. It is a general curve renderer and as such cannot work with hinting or similiar techniques.

`StrikeFont` may can contain artifacts and may not be optimized for the current display, because it is prerendered. It depends on the techniques that were used while creating the bitmaps. How similiar the display hardware of the system the font was prerendered on is to the system of the user may also have an impact. If the prerendering is very good, this font may have at its optimal sizes the same quality as the other approaches. But it cannot scale as well as the other two can with vector fonts. The simpleness however also makes it very fast.

4.5.3 Interactivity

Both Pango and the `StrikeFont` approach draw directly onto a canvas. The Balloon Renderer on the other hand renders each glyph as its own object. The glyphs are placed in a container that has a basic layouting logic. The container forces a new line if the next glyph object would no longer fit on the line.

The glyphs can be inspected and interacted with like other objects in the system, as can be seen by the selected glyph object in Figure 4.17. This allows for new ways of interaction. One specific use case for this is part of the chapter about live editing [22].

4.6 Proposed Alterations

At the moment, text layouting is done by a composer and font object under the assumption that everything can be rendered context-free. The font interface includes a method to determine the length of a given text, but the composer also has functions to do this which are under the assumption that each input character has one definite width. Neither the composer or the font have the logic required for complex text rendering.

What is missing is a shaper object. The shaper turns its input into glyphs with positions based on the given font. This enables rendering with typographic features. A shaper should be able to handle text of any length, but it would also be enough if the shaper is asked for information on a word basis. The splitting of text into words could be done by the composer. On a conceptual level, the shaper has to resolve the function

$$f(font, size, direction, features, script, language, text, context)$$

into a list of glyphs and their positions.

To reduce changes to the interaction between composer and font object, the shaper could be used by the font to answer the composer. All information that the font cannot provide on its own would be be forwarded as a query to the shaper of the font. Methods that determine the length of a text for example could only be resolved with help of the shaper. The shaper object and its logic could be completely implemented in Squeak or it could make use of an external plugin such as Pango as a backend.

One way to integrate the shaper would be to give every font one. This would mean that shapers for existing fonts would have to be created. For example, `StrikeFont` would require a very simple shaper that just places glyphs beside each other with the only possible additional feature being kerning. Each font could have a function that returns it assigned shaper class. Introducing a shaper for all fonts creates a general text rendering pipeline all fonts support, however it also requires rewriting and integrating shapers for functionality that already works. The interfaces of font and composer should be changed according to this. The composer cannot provide certain functionality such as determing the length of a text without the font and its shaper.

Another way to integrate shapers is by creating a new subclass of `AbstractFont`, a `ShaperFont`. Existing fonts that do not require shapers could subclass `AbstractFont` directly and work as before, maybe with slight alterations to their interface. The new fonts that need shapers could subclass from the new class and define their own shapers. The interface of font and composer also has to be altered here, but it could be possible to only change `ShaperFont`. Also maybe a new subclass of composer that is for fonts with shapers could be introduces, because this would also lessen changes to the existing objects and their interfaces.

Other parts of the font interface would also have to be changed according to this. At the moment `AbstractFont` defines a function `#widthOf: char` which assumes each input character has a specific length. This is true if there is a delimeter before and after it, bu not if it has context. This means this function should not be used for layouting and similiar tasks. It is sometimes called to provide information on specific characters such as 'space', 'x', 'm' and 'i'. These special cases should have there own functions in the font object. They usually have a specific purpose, for example the 'i' is assumed to provide a small width. Since the designer can give the ‘i’ any width they want, this can lead to unwanted results. If it has its own function with a stated purpose, the font can provide according information.

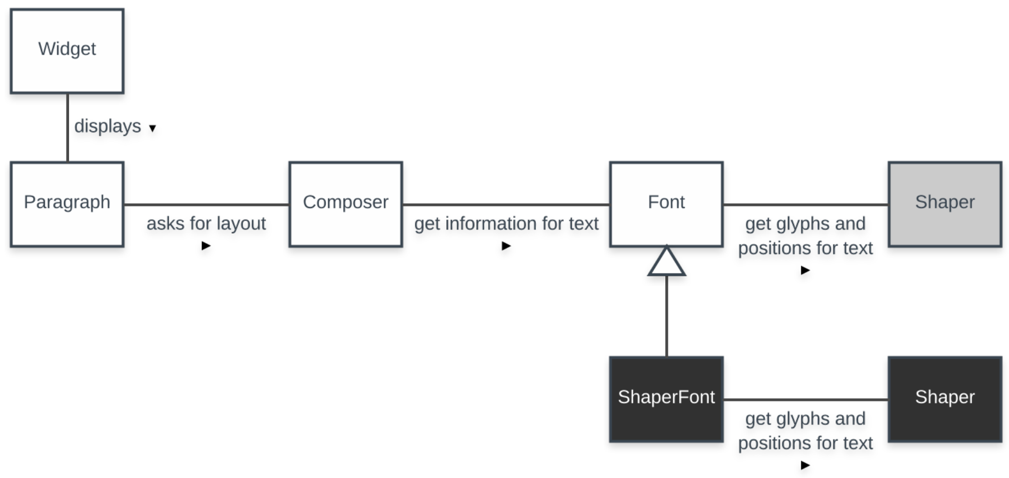

Widget
displays ▾
Paragraph
asks for layout ▸
Composer
get information for text ▸
Font
get glyphs and positions for text ▸
Shaper
ShaperFont
get glyphs and positions for text ▸
Shaper

Figure 4.18: The UML class diagram shows the main objects which are relevant for the current text rendering process in Squeak and two ways to add the alteration. The objects which are already part of the rendering process, a widget, a paragraph, a composer, and a font, are in white boxes. The object which the alteration introduces is a shaper. All fonts could use a shaper object, in this case the grey box would be added. To reduce changes to the already existing fonts it would also be possible to introduce a font subclass which uses a shaper, in this case the dark boxes would be added.

4.7 Discussion

All approaches are usable depending on the use case. `StrikeFont` does not support complex rendering, but is completely integrated into the live environment and very fast. It cannot render all text in a comprehensible way, but it can render all ASCII-characters which is enough for basic English text. It also supports a few other scripts that do not necessarily need typographic features. Pango is already implemented and has several years of work in it. But it also has its own code and because of that has to be interacted with as an external component. The Balloon renderer approach lacks most of the layouting logic and cannot process the neccessary information for complex text rendering on its own, but it provides the text elements as objects.

Which font files should be supported can also alter the use case. The Balloon renderer for example needs a .ufo file, but not every font designer or type foundry wants to provide their font in a non-binary format. Some are just used to other formats and fear information loss from converting.

A live system should be able to support complex text rendering. It is needed to support text on an international level and for aesthetic reasons. This does not necessarily mean removing simple text rendering approaches, it means the general text rendering process should also allow complex text rendering.

The proposed alteration would likely help creating such a process. Having a shaper as part of the rendering pipeline allows typographic features which are very important for international scripts. Implementing a shaper, without using an external shaper such as Pango, would still not be easier, but would be possible.

The comparison was heavily based on implementations in Squeak, but the concepts can be applied to other live environments. They should provide the possibility to have complex text rendering which means they have to include font and shaper in their text rendering process. This interface should be general enough that both mostly self-supported approaches like `StrikeFont` in Squeak or external solutions like Pango can work with it.

4.8 Related Work

There are several already implemented modules that cover text rendering or parts of it[12]. This provides further knowledge about text rendering processes and what problems are still open. This makes the challenges of complex rendering clearer and helps with implementations of new text rendering approaches.

It also helps with integrating already implemented solutions into systems, for example information on Pango can be found in papers specifically about it [38]. Understanding how Pango is structured and how it works allows using the parts that are currently needed. It is also a good example of the functionalities a shaper has to provide.

General information about fonts file and typography are also useful for understanding text rendering. Knowing what certain scripts need in order to be comprehensible helps understanding complex text rendering. General information about the topic typography can be found [13]. For example, understanding what a ligature is and what it is used for makes it clearer what complex rendering should provide.

Understanding live environments and how they work is also important for creating new ways to render text in them. There is information on Smalltalk, which Squeak is based on [14] and also on Squeak itself and its background [17]. This contains its basic principles including how `BitBlt` works. Another system that is based on Smalltalk for example is Pharo[13] which has its own text rendering approaches.

[12]State of Text Rendering: `http://behdad.org/text/`, visited on 2017-07-27.
[13]Pharo: `http://pharo.org/`, visited on 2017-07-27.

4.9 Conclusion and Outlook

In this chapter we looked at different ways of rendering text in a live environment. The intention was to not only have simple text rendering, but complex text rendering for international text. We compared the rendering approaches on whether they allow correct complex rendering of fonts, the supported quality of the result and if they are compatible with a certain kind of interactivity that live environments offer. At the moment the approaches are made for special cases, the external Pango plugin was used because of its support of complex text rendering, while the focus of the Balloon renderer was to have objects for glyphs in a text. One default way of text rendering in Squeak is using a bitmap font which does not fulfill the wanted criteria, but is very fast and simple. Based on our experience with the default rendering process, we propose an alteration which should add support for complex text rendering, or at least most typographic features, to the environment. Then we discussed this change and how these concepts apply to other live environments. The basic involved entities are the same in all environments and for internationalization it is best to directly support complex rendering in the default text rendering process.

Future work could contain an implementation and testing of the alteration proposed in this chapter. The current solution has a focus on rendering words based on typographic features correctly, but the other factors could also be looked at in more detail, for example if all directions that text can have are supported. Related to this, splitting into correct words and parts, which can then be given to the shaper, based on delimeters, script and language can be looked at. The encoding of text in Squeak and other environments could also researched and maybe altered, as at the moment there are special cases, for example determining the length of text in Japanese has its own function because of encoding. Also the current solution for having text elements as objects for more interactivity is not compatible with the standard text rendering, it would be interesting to find out if an integration is possible.

5 Live Glyph Editing in a Live Environment

Designing fonts is a very visual process, and requires a lot of visual feedback and testing; however, many steps are required to get visual feedback. We explain the differences between modern font formats with focus on font source data, and illustrate the steps to get visual feedback in the current design process. To make the feedback process faster, we propose a system that allows editing and displaying the font from within the system, based on the font source format. As proof of concept, we present a prototypical implementation using Squeak/Smalltalk as our system. Based on our implementation, we evaluate and discuss the feasibility of our approach.

5.1 Introduction

Digital fonts are everywhere: in e-books, browsers, office software, smartphone apps, or even hard copies. Nowadays, a world without digital fonts is hardly imaginable, the availability of fonts is taken for granted. However, making good fonts is a hard, complicated, and lengthy process.

A reason for the hard process is the need for visual feedback. It is a long way from designing a font to being able to see and it in an appropriate environment. Several steps need to be done, for example compiling the font, installing it on the system, or setting up an appropriate test scenario. This test scenario can be anything the font is going to be used for, such as a Microsoft Word document with selected characters.

In this chapter, we explore a solution that simplifies the process of getting visual feedback and propose a system that offers such live visual feedback for editing glyphs. Our approach is to use an object-oriented live system. With the font and its glyphs as objects, we make it possible to edit individual glyphs. For that, we need an glyph editor inside the system. Also, we propose an own text component that is capable of displaying the glyph objects. This way, we are able to show glyph changes live, as soon as they occur.

Section 2 imparts background knowledge that is useful for understanding this chapter. The next section describes the approach we use for the project. In section 4, we focus on an exemplary implementation using the Squeak/ Smalltalk system. Section 5 describes how we evaluate the project, discusses the measured values, and points out limitations. In section 6, we present previous work that relates to this chapter. The last section concludes the chapter.

5.2 Context

Posters, mockups, or websites are very visual products. So are fonts too. When designing such visual products, it is important for a smooth design process to get fast, preferably even live visual feedback for changes, to see the impact those changes have. When talking about *live visual feedback* for changes, we mean direct, immediate updates of the relevant graphical user interfaces, within half a second after the change occurs at the maximum. Without such live feedback, the amount of time actually spent with designing—and with that the productivity—decreases.

Our project aims to make visual feedback for font design faster. To achieve that, we use an object-oriented, self-supporting live environment, namely Squeak [18]. An advantage of Squeak is the concept of a shared object space. Everything in Squeak is considered an object, including fonts. The shared object space allows to access and change every object of the system, from within the system.

5.2.1 Bitmap Fonts versus Vector Fonts

Historically, the so called bitmap fonts were very popular. Conceptionally, a bitmap font is a collection of pre-rendered bitmaps, each corresponding to a single character. Since the bitmaps can be copied onto the screen without further steps, they have a really good rendering speed. However, bitmap fonts have downsides, too. For example, scaling bitmaps usually leads to a loss of visual quality. As a result, a bitmap font is rather a collection of bitmap fonts, each in a fixed point size. Nowadays, there are a lot of different display sizes and resolutions, making it unfeasible to pre-render every possible point size that might be needed. The main reason for the unfeasibility is not the increased amount of memory needed, but rather the additional design process. Every point size of a bitmap font needs to be designed and optimized individually to achieve the optimal visual quality.

A solution for the scalability problem are vector fonts. Similar to vector graphics like SVG, the individual characters (called "glyphs") are built out of certain graphical primitives. These primitives are lines, conic bézier curves, and cubic bézier curves. A bézier curve has a start point, an end point, and one (conic) or two (cubic) control points, determining its pathway. A line can be seen as a special case of a bézier curve. We show the primitives in Figure 5.1.

When multiple bézier curves are attached to each other (each end point attached to the start point of the next curve), they yield a contour. Since the last end point is connected with the first start point, contours are closed. Contours can also be seen as areas, specified by bézier curves, so called bézier patches. A single glyph might have multiple contours. All contours together result in an outline. Refer to Figure 5.2 for an exemplary outline of the latin character "O".

Vector fonts have the concept of source and binary versions. Similar to certain programming languages, the compilation is performed for performance and opti-

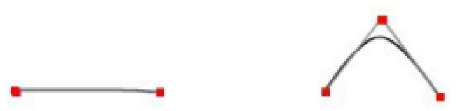
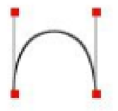



Figure 5.1: The three primitives, from left to right: line, conic bézier curve, cubic bézier curve.

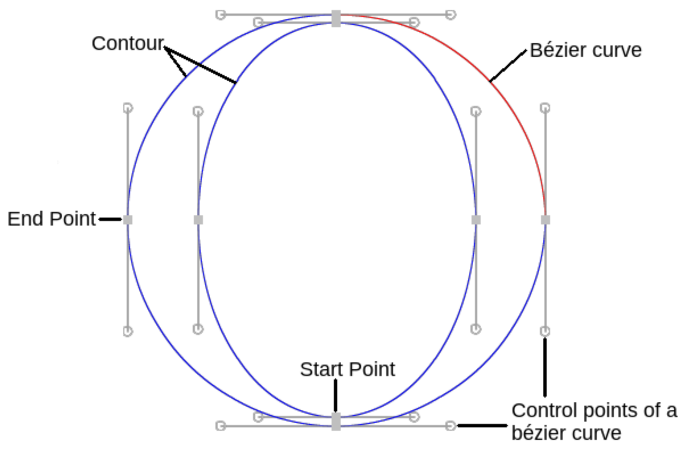


Figure 5.2: Outline of the letter O of Open Baskerville, with annotations.

mization reasons. A common font compilation tool is fontmake[1]. While the binary formats are optimized for end usage, source formats have their advantages too. For example, source formats contain unoptimized information, which are easier to understand for humans, and are before anything else easier to edit. For this reason, the actual font design is mostly done using source formats.

The source format we use for this project is the Unified Font Object format, version 3 (UFO3)[2]. UFO3 is a folder based font format. Each glyph is stored in its own file, in a folder called "glyphs". The UFO folder contains additional files, each describing certain properties. Considering the structure of UFO, it is simple to transfer those UFO files into an object-oriented representation. This representation allows us to work object-oriented within our system. Moreover, following an open standard like UFO allows exporting and usage of such fonts outside of the system.

5.2.2 Motivation

Consider the following scenario: You as a software developer want to develop a small game. While designing the main menu, you notice that none of the system fonts really fits into the style and atmosphere of the game. So you open an external editor like Glyphsapp[3] and start designing outlines for your own font. When you have designed a the first version, you export it as an OTF file, the binary OpenType Font format. Then you need to integrate the OTF file into your system. Only then you get visual feedback and can assess the quality of your work. This current process of getting visual feedback is illustrated in Figure 5.3. Since you have no prior experience with font design, the letters do not harmonize well at first try. This can be seen in Figure 5.4. For a second version, you need to go through all the steps above once again, which is cumbersome. Ideally, the process would consist of only one step, to spend more time with the actual design.

This project explores the possibility to edit fonts within your system and get immediate visual feedback for your changes. Vector fonts consist of more than just glyphs, such as ligatures and localizations. However, for the given scenario, those complex font features are not needed, it is not even necessary to design a complete character set. For the mock of the main menu shown in Figure 5.4, only 13 glyphs are needed. Therefore, being able to edit glyphs is completely sufficient for this project.

5.3 Concept

The very basic idea of this project is simple: Even though OpenType fonts are the current state of the art, they introduce a lot of intermediate steps into the design

[1]Font make: `https://github.com/googlei18n/fontmake`, visited on 2017-07-30.
[2]UFO3 specification: `http://unifiedfontobject.org/versions/ufo3/`, visited on 2017-07-30.
[3]`https://glyphsapp.com/`, visited on 2017-07-30.

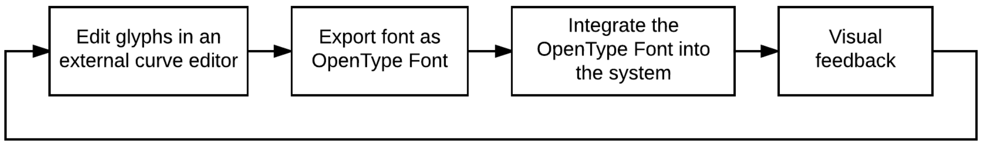

Edit glyphs in an external curve editor
Export font as OpenType Font
Integrate the OpenType Font into the system
Visual feedback

Figure 5.3: The current process font designers need to go through to get visual feedback

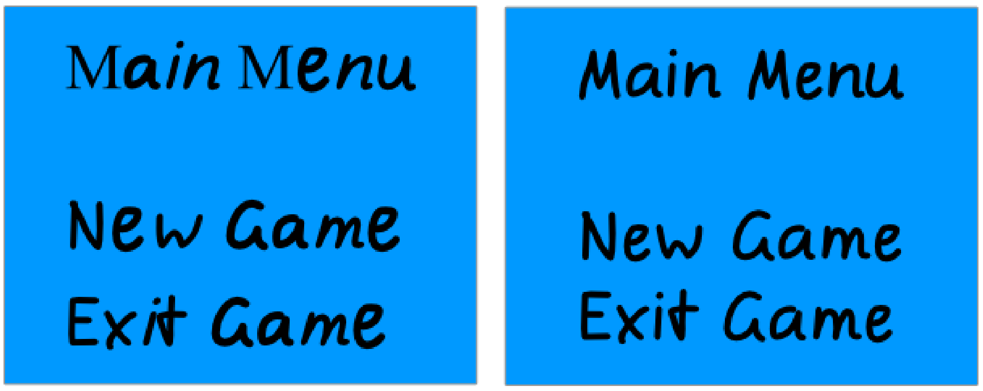


Figure 5.4: Mockup of a the main menu. The left side shows the first version, the right side shows how it should look like.

process. To avoid these intermediate steps, we directly change and render the source format (UFO). This brings three challenges: First of all, we need a component to render glyphs from source. Second, we must be able to edit glyphs from within the system. Third, during editing, we need a way to propagate the changes to the rest of the system.

In the UFO format, each glyph outline is stored in an external file. For a start, we need a parser, to turn the file contents into an internal object representation. This data based glyph representation must be sufficient for both the renderer to draw a glyph, and the glyph editor to build a visual representation of the glyph. We also introduce an object representation of an UFO font, which knows all its related glyph objects.

5.3.1 Rendering Glyphs

For rendering glyphs, we introduce two new components. First, we need a component to display strings. Basically, this component is an user interface element. The component consists of a text to display and a font to use. Second, we need a component that can render our glyph objects. The displaying component uses the rendering component to actually display the string.

To display a given string, we first need to get the collection of glyph names corresponding to the string. To find out the glyph names, we iterate over the given string and consider the unicode position of each letter. For each letter, we iterate over all glyph objects of our font, and check whether the glyph object's unicode matches the letter's unicode. If yes, we found the corresponding glyph name. This process of turning text to corresponding glyph names is called shaping.

The rendering component then iterates over the glyph names. For each glyph name, it looks up the corresponding glyph object of the font. The glyph's outline is a set of concatenated bézier curves, which can be seen as bézier patches. These bézier patches are directly rendered by the system. Being able to render bézier patches is a requirement for the system we use. At last, the offset for the next character advances by the width of the just rendered glyph.

5.3.2 Editing Glyphs

In the context of vector fonts, editing a glyph means editing a glyph's outline. Outlines are composed of bézier curves. So for editing a glyph, we basically need a bézier curve editor. In Figure 5.5 we show how such an bézier curve editor could look like.

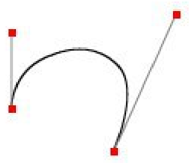


Figure 5.5: Mockup of a bézier curve editor inside the live system, showing a cubic bézier curve. The control points are on the top, start and end point on the bottom. The actual bézier curve is rendered in black, the lines to the control points in gray.

The visual representation of the glyph is built from the outline of the glyph object. The editing abilities of the editor are limited to moving control points. Changes are directly written back into the glyph object.

For a full scale editor, we would also need functionality to add or remove bézier curves, or contours. However, for this project, editing existing curves is sufficient, if we change an existing font rather than creating a completely new one.

5.3.3 Making visual feedback instant

While editing a glyph, the changes made to it need to be propagated to the whole system. To propagate the changes, and thereby also update all other occurences of the modified glyph, all related displaying components must be updated.

While editing a glyph, all changes are immediately integrated into both the glyph representation and the font object (via its glyphs). Therefore, the displaying component's font is already up to date. The only thing left to do is rendering the displaying component once again.

But how do the displaying components actually learn that there are changes? There are two different approaches: timer-based, and event-based.

5.3.3.1 Timer-based

The basic idea is simple: Give the font a flag, indicating whether it was modified. All displaying components check in regular time intervals whether the flag is set. In case it is, they need to rerender. Finally, in the same regular time interval, the flag needs to be unset.

Instead of using a binary flag, it is also possible to use timestamps. For that, both the font and the displaying component need a timestamp. The font needs to store the time of the last modification, whereas the displaying component stores the time of the last font update. If the last modified timestamp is newer than the last updated timestamp, the displaying component has to rerender and update the last updated timestamp. If the timestamp comparison takes too long, we can use a font version number instead. The font version number is basically an integer that is incremented every time a change to the font is made.

The second variant is superior to the first one. For example, the second variant allows different updating intervals. In case we lack performance, it would be possible to give some displaying components priority and let them poll frequently. Meanwhile, less important displaying components poll less often and save thereby performance, but are still capable to recognize that there is a new version of the font.

5.3.3.2 Event-based

The event-based approach is inverse to the timer-based one: Instead of periodically checking the font for changes, it is the font that notifies the text morphs as soon as changes occur.

Many self-supporting environments come with the ability to get a list of all instances of a certain type. For a first simple way, this is enough to find out which displaying components need to be rerendered. Whenever a glyph is edited, the glyph object reports it to the font, which iterates over all displaying components and tells them to rerender. As a performance improvement, it is possible to rerender only those displaying components that actually use the font that changed.

However, this implementation still has a lot of room for improvement. For example, if someone introduces another displaying component that uses our rendering component, we need to update those displaying components too. But to do that,

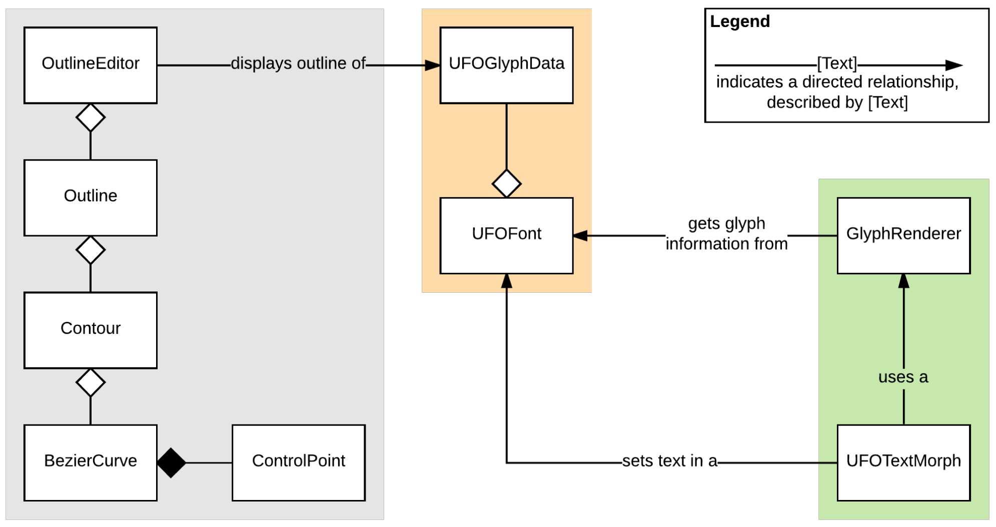


Figure 5.6: UML class diagram of the whole system. The classes are put into one of the following categories: either data structure (orange), rendering (green), or editing (gray).

we have to change the source code of the font object. Moreover, the font needs to know how to rerender the targets, or at least what message they expect to receive.

For a better solution, it is possible to use the observer pattern. The font acts as model. When the font of a displaying component is set, it registers itself as an observer of the font. Everytime a glyph is changed, the font notifies all its observers via a well-defined interface. The observers can decide on their own how to deal with the update—in our case, they rerender.

5.4 Implementation

During the project, we developed a protoypical implementation of the concept described in the previous section. As a self-supporting live system we used Squeak, a Smalltalk dialect. All code listings refer to the Squeak trunk, which is currently at version 6.0alpha.[4]

5.4.1 Architecture

Refer to Figure 5.6 for an overview sketch of the whole system.

[4]Squeak/Smalltalk programming system: `http://files.squeak.org/6.0alpha/`, visited on 2017-07-30.

There are three major components worth mentioning: The data structures (orange), the editor functionality (gray), and the rendering (green).

5.4.1.1 Internal UFO representation

Listing 5.1: An exemplary glif file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<glyph name="a" format="1">
  <advance width="447"/>
  <unicode hex="0061"/>
  <outline>
    [...] <!-- just some more contours -->
    <contour>
      <point x="188" y="6" type="curve" smooth="yes"/>
      <point x="122" y="6"/>
      <point x="117" y="51"/>
      <point x="117" y="91" type="curve" smooth="yes"/>
      <point x="117" y="175"/>
      <point x="226" y="198"/>
      <point x="287" y="236" type="curve"/>
      <point x="287" y="145" type="line" smooth="yes"/>
      <point x="287" y="68"/>
      <point x="270" y="6"/>
    </contour>
  </outline>
</glyph>
```

UFO3 is a folder based file format. Each glyph is stored in its own ".glif" file. Those ".glif" files use XML. Refer to Listing 5.1 for an excerpt of Open Baskerville's glyph "a"[5]. Such a glif file is represented by an `UFOGlyphData` object. The `UFOGlyphData` stores the glif file contents. Since the glif file is an XML document, it is standing to reason to store it as an `XMLDocument`. We generate the `XMLDocument` using `XMLDOMParser parseDocumentFrom: aStream`.

For a font object representation , we introduce the class `UFOFont`. An `UFOFont` has to know all its glyphs. The rendering component will query the `UFOFont` for specific glyphs. Therefore, we use a `Dictionary` as data structure to store the glyphs: with the glyph's name as key, the `UFOGlyphData` as value. This allows fast access to a particular glyph.

5.4.1.2 Rendering glyphs

For the displaying component, we introduce the `UFOTextMorph`, which has the same use cases a `StringMorph` has in Squeak. There will be occurences of glyphs that are not in an `UFOTextMorph`. An example are the contents of a text field. However, the

[5]Open Baskerville: `https://github.com/klepas/open-baskerville/blob/master/OpenBaskerville.ufo/glyphs/a.glif`, visited on 2017-07-30.

principle is the same: There is a string to display and a font to use. Therefore, we can abstract from those and use the `UFOTextMorph` as representative.

To display glyphs, we introduce the class `GlyphRenderer` as the rendering component. The `GlyphRenderer` belongs to a certain `UFOFont` and can render the font's glyphs. To display the glyphs, the `GlyphRenderer` uses a `BalloonCanvas`, which has the possibility to draw bézier shapes and therefore fits out needs.

Listing 5.2: Drawing bézier shapes onto a BalloonCanvas

```
 1 GlyphRenderer>> drawGlyphShape: aContourCollection
 2   scaleFactor: aNumber on: aCanvas at: aPoint color: aColor
 3 
 4   aCanvas asBalloonCanvas preserveStateDuring: [:canvas |
 5     canvas
 6       aaLevel: 4;
 7       transformBy: (MatrixTransform2x3 withOffset: aPoint + (0 @ font ascender *
            ↪ aNumber));
 8       transformBy: (MatrixTransform2x3 withScale: 1 @ -1 * aNumber);
 9       drawGeneralBezierShape: aContourCollection
10         color: aColor borderWidth: 0 borderColor: nil]
```

The Listing 5.2 shows how our renderer works. In this drawing call, line 7 and 8 are worth a closer look. In line 8, we perform two steps in one call: First, we scale with `aNumber`. The value of `aNumber` is desired point size divided by units per Em. Units per Em is a property of the font and therefore stored in the font object. As a result, we perform with this step the scaling to the desired point size. Second, we invert the y-axis, and thereby transform the coordinate system. This transformation is required, since the bézier shapes and the `BalloonCanvas` have different origins: for the bézier shapes, the origin is the bottom left corner, for the BallonCanvas, it is the top left corner. As a result of inverting the y-axis, the bézier shapes have now the same axis orientation, but will be displayed at the wrong place. This is fixed in line 7, by offsetting all shapes with the scaled ascender. The ascender of a font describes the height that lowercase letters go above the x-height, and is stored in the font object.

5.4.1.3 Editing glyphs

Editing glyphs requires a visual, touchable representation of the glyph object. We use the Morphic framework of Squeak [25] to build such a representation. Glyphs have an outline, which consists out of contours. Contours are composed of bézier curves. Bézier curves have control points. In our implementation, we stick to this hierarchy, and introduce a class for each element. All these classes inherit from `Morph`. Each class has the objects of the next lower hierarchy level as submorphs. As a top level morph, we introduce the class `OutlineEditor`. This `OutlineEditor` has a glyph object, and builds up the described morph structure based on the glyph object. The editor can be seen in Figure 5.7.

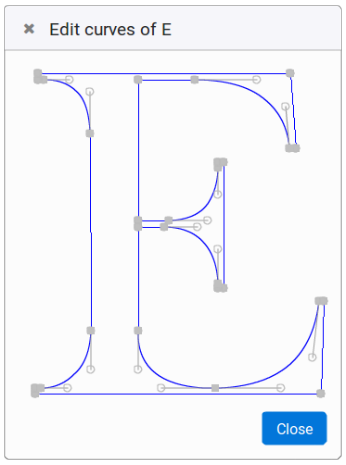


Figure 5.7: The bézier curve editor we built inside the live system. Currently, the outline of the latin letter "E" is opened for editing.

Although talking about editing, for obvious reasons the outline must be rendered as well, to make editing possible. The actual rendering of the outline to edit is performed in `BezierCurve`. A `BezierCurve` knows the coordinates of its control points, more is not necessary. The render call is similar to the one described above and therefore not further discussed.

The `ControlPoint` class is responsible for the actual editing of the outline. A control point is draggable and listens to mouse-move events. Whenever moved, it sends a message to its owner, a bézier curve, that it needs to rerender. Also, we need to update the corresponding `UFOGlyphData` by updating the control point coordinates. To update the glyph object, we go up in the hierarchy up to the `OutlineEditor` and inform it about the change. The `OutlineEditor` then serializes the `Outline` to XML and updates the glyph object.

5.4.2 Propagating changes

After editing a glyph, the changes to its bézier curves need to be propagated. We discussed different approaches in the previous section. For our implementation, we chose the event-based approach, using the observer pattern. The observer pattern has two roles, model and observer. As model, we use the `UFOFont`, which informs its observers about changes to the font. An observer can be any graphical element that uses the `GlyphRenderer`, for example an `UFOTextMorph`.

The Squeak system has the observer pattern already built in: Every `Object` has a collection of dependents. Sending `changed` to `Object` will send `update:` to each of its dependents. Therefore, the application is as simple as shown in Listing 5.3.

`UFOGlyphData>>updateOutline:` is the message that is sent while editing a glyph. So whenever a glyph is edited, the glyph object tells its font that it was changed, and triggers thereby the process of propagating the changes. In Listing 5.3 we show the

Listing 5.3: The implementation of the observer pattern

```
UFOTextMorph class>>newWithFont: anUFOFont

  ↑ self new
      font: anUFOFont

UFOTextMorph>>font: anUFOFont

  font := anUFOFont.
  font addDependent: self

UFOTextMorph>>update: anUnusedParameter

  self changed "this triggers a rerendering"

UFOGlyphData>>updateOutline: anXMLDocument

  self outline: anOutlineXMLDocument.
  self font changed
```

implementation exemplary for the `UFOTextMorph`, but it works as well for any other graphical element.

5.5 Evaluation and Discussion

In this section, we reflect and evaluate the project. For that, we first define measurement parameters and the system the tests are run on. As second step, we perform the actual measurements. At last, we take limitations of the implementation into consideration.

5.5.1 Measurement parameters

The goal of the project is to get visual feedback as fast as possible, preferably live. Therefore, we measure the time it takes to apply the changes, from the time the change is made till all occurences are updated, the *compile time*. We split this compile time in two major categories.

First, there is the time needed to propagate the changes. This step is composed of two parts: Serializing the changes of the editor back into the `UFOGlyphData`, and getting a list of all instances that need to be updated, and inform those instances about the new version.

Listing 5.4: Measurement of the time needed for updating the glyph object and propagating changes. We simulate a scenario with 20 morphs as observers.

```
observers := OrderedCollection new.
20 timesRepeat: [observers add: Morph new].
outline := Outline for: <XMLDocument>.
```

```
editor := outline editor.

[1000 timesRepeat: [ editor glyph updateOutline: outline xmlElement.
  observers do: [:each | each changed]]] timeToRun
```

The `Outline` does not use a cache, so each sending of `xmlElement` to it will rebuild the XML from scratch. Thus, there is no need to actually change the outline. As for `observers`, we use a list of morphs to simulate the `UFOTextMorphs`. This simplification is valid too, since the observer pattern used in the previous chapter grants access to the relevant objects as easy as this.

The 20 is a magic value and stands for the number of `UFOTextMorph` objects that we can expect to have in the system at once. We perform the test 1000 times, in order to get a more representative value for the average time needed.

Second, the changes need to be applied. Basically, this means rerendering. For the exact scenario, we would need to render 20 test strings once onto a canvas. We use a simplification and render a test string 20 times onto a canvas. This simplification is valid, since the total amount of morphs to render is unchanged. This test is performed 1000 times too, for a more representative average time.

Listing 5.5: Measurement of the time needed for rendering

```
testString := 'Lorem ipsum dolor sit amet'.
glyphNames := Shaper new shapeString: testString.
form := Form extent: 100@50 depth: 32.
canvas := FormCanvas on: form.

[1000 timesRepeat: [
  20 timesRepeat: [
    renderer drawGlyphs: glyphNames size: 16
      on: canvas at: 0@0 color: Color black]]] timeToRun
```

`renderer` is an instance of `GlyphRenderer`. Its initialization is here skipped, as it introduces unneccessary complexity and is not relevant for this measurement. All tests were run with Open Baskerville[6].

5.5.2 System specs

All tests were run with on a computer with the following specs:

Hardware	
CPU	Quad-Core Intel® Core™ i5-4690 CPU @ 3.50GHz
RAM size	8,1 GB
Software	
Squeak image version	Squeak6.0alpha (32 bit)
VM version	Closure Cog[Spur] VM [CoInterpreterPrimitives VMMaker.oscog-cb.1919]
OS version	elementary OS 0.4.1 Loki

[6]Open Baskerville: https://github.com/klepas/open-baskerville, visited on 2017-07-30.

5.5.3 Discussion of the measured values

It took 50ms to execute Listing 5.4. Since the listing runs the serialization and propagation 1000 times, a single run takes about 0.05ms. We consider this fast enough for live feedback.

The runtime of Listing 5.5 is way higher. For rendering 20 morphs 1000 times, 288,126 ms were needed. This is an average time of 288ms for rendering 20 morphs. The average rendering duration of a single morph is 14.4ms. With an average time of 288ms, the current implementation might be too slow to present instant feedback on moving control points.

However, there is room for further improvement. For example, font designers work mainly with cubic bézier curves. The `BalloonCanvas` we use for rendering, however, is working with conic bézier curves. As a result, the glyph outlines are converted from cubic to conic bézier curves for each render call. Introducing a cache for the conic curves might speed up the render call. Moreover, the current implementation triggers rerendering with every mouse-move event. To save performance, it is possible to trigger the rerendering only when mouse-up events occur. This way, rerendering is needed less often, but the visual feedback still happens fast enough.

5.5.4 Limitations

The implementation has some limitations though. For example, there is an issue with the current renderer implementation. An outline can contain multiple contours. If those contours overlap each other (as shown on the left side of Figure 5.8), the rendering result has cutouts (as shown on the middle).

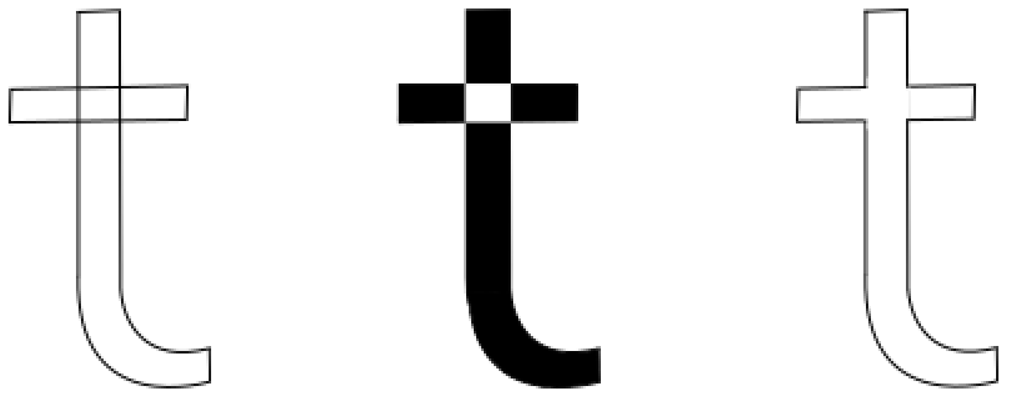


Figure 5.8: In the left: outline of the glyph "t", as designed. In the middle: rendered outline with a cutout where the contours cross. In the right: outline of the glyph "t", with bézier curves unified.

To avoid such defects, the overlapping parts need to be removed. This can be achieved by unifying both contours. The result of such a unification is shown on the right side of Figure 5.8 . Usually, tools like fontmake[7] perform this step, when they convert the font to the binary OTF format. Since we directly render from the sources, we never reach the normal compile time. Instead, the union has to be done by the renderer too.

There is one more limitation of the current implementation. For this project, we focused on editing glyphs. We ignored more complex font features like ligatures, since they are nice to have, but not mandatory. However, this is not entirely true. There are writing systems that have complex, context sensitive rules. For example, in the arabic writing system, letters may change their shape, depending on whether they are at the start, in the middle, or the end of a word. In devanagari, the indian writing system, it occurs that multiple letters are joined and result in a single glyph. To achieve such behaviour, we need to support according rules, and a shaper component. The task of a shaper is to deliver a sequence of glyph names for a given input string. In Listing 5.6 we show how our project performs shaping.

Listing 5.6: Very basic implementation of a shaping component. For each letter, we search for a glyph with the same unicode as the letter.

```
Shaper>>shapeString: aString

  ↑ aString collect: [:letter |
      self font glyphs
        detect: [:glyph | glyph unicode = letter unicode ]
        ifNone: [self undefinedGlyph]] as: OrderedCollection
```

However, a letter-based implementation like this will not work for complex writing systems. This has multiple reasons. First, not every glyph has a unicode. Second, when considering only one glyph, we cannot get enough context for more complex rules.

Fortunately, there are already shapers that can deal with those complex writing systems. An example for a current shaper is HarfBuzz, which is also used in Firefox, Chrome, or LibreOffice [2]. Unfortunately, HarfBuzz works with the OpenType Format. So to enable our project to support complex writing systems, we need an HarfBuzz backend that is working with the UFO format, and a plugin for squeak to communicate with that backend.

[7]Font make: `https://github.com/googlei18n/fontmake`, visited on 2017-07-30.

5.6 Related Work

There are many other products that depend on visual feedback, too. In this section we look at projects or ideas with a concept similar to ours.

5.6.1 Inventing on principle

In 2012, Bret Victor gave a keynote at the CUSEC[8]. His talk was called "Inventing on principle" [41]. During the talk, he stresses the importance and the need to get live feedback when designing visual products, and supports his remarks with illustrative examples. Although his examples refer to computer graphical topics (including games) rather than fonts, the basic idea and motivation is very similar to the one of this project.

5.6.2 Direct manipulation

Another interesting idea is the concept of direct manipulation as proposed by Shneiderman and Plaisant. By the example of the history of word processors, they explain the development towards the concept of direct manipulation: In the beginning, editing files was done line by line, with line-oriented command languages. To make changes or switch the current line, commands had to be typed. Over time, this has developed to what-you-see-is-what-you-get (WYSIWYG) editors like Microsoft Word. Those have numerous advantages, making interaction easier: Seeing a full page of text, seeing the document without formatting commands and thus increasing readability, cursor interaction, or the immediate display of the results of actions like centering text [35, pp.194]. According to the authors, direct manipulation is based on three principles:

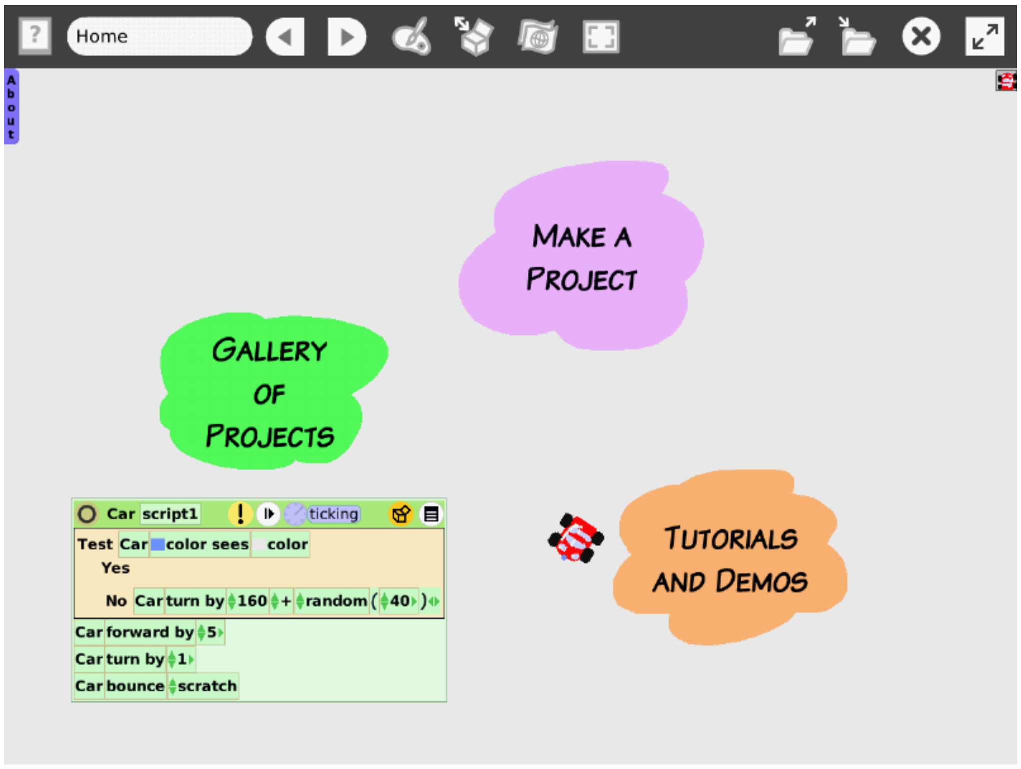
1. Continuous representations of the objects and actions of interest with meaningful visual metaphors.
2. Physical actions or presses of labeled buttons, instead of complex syntax.
3. Rapid, incremental reversible actions whose effects on the objects of interest are visible immediately [35, p.214].

During our project, we have followed those principles to make editing glyphs live and easy to use.

5.6.3 Etoys

"Etoys is an educational tool for teaching children powerful ideas in compelling ways" [9]. This is an extract of the description on the official web page. While

[8]The Canadian University Software Engineering Conference: `http://2012.cusec.net`, visited on 2017-07-30.



```
Car script1 ticking
Test Car color sees color
  Yes
  No Car turn by 160 + random ( 40 )
Car forward by 5
Car turn by 1
Car bounce scratch
```

Figure 5.9: Screenshot showing the home screen of Etoys. The script in the left bottom corner controls the car. Every change to the script directly applies to the behaviour of the car.

learning new things, children can get some first experience with programming through the scripting abilities of Etoys. For example, Figure 5.9 shows the home screen of Etoys. The car is driving on the screen, controlled by the script in the bottom left corner. When making a change to the script, it instantly affects the behaviour of the car. This live feedback makes learning to program a lot easier, because children can experiment with the code and directly see how their changes work out, while doing the changes.

5.7 Conclusion

In this chapter, we have proposed and presented a way to edit glyphs from within the system and get live visual feedback. We have discussed our approach to use the font source format for both editing and rendering glyphs, and pointed out the challenges that are entailed in this approach. We have illustrated how editing and rendering glyphs could work, as well as propagating the changes made to the glyph. As a proof of concept, we have developed a prototype that implements our approach for live feedback, using Squeak/Smalltalk as a live environment. To evaluate the liveness of our prototype, we have measured the average time needed to update the system after editing a glyph. We discussed those updating times, found them as not perfect, but good enough, and suggested further perfomance optimization strategies. Moreover, we have considered technical limitations of our prototype, and how these limitations can be addressed.

With this prototype, we have proposed a way to make the visual feedback loop for fonts faster. We think it could speed up the process of creating fonts. We expect it to be especially helpful when the font needs to fit into a certain graphic style, as it has to when designing a game. We hope it encourages more developers to consider making a custom font.

6 Considerate Code Generation in Live User Interface Design

There are many tools to live-edit the state of running applications. However, edits are usually not persisted in code and will be lost upon restarting the application. We propose the concept of a live-tracking system that finds suitable places in the application's source code to apply the changes of arbitrary live-editing tool support. This system will support developers to stay within the context of their running application without the need to switch back to their programming environment for every change. We examine challenges that we encountered while implementing a proof-of-concept, for example how to optimize memory usage of the live-tracking system and discuss consequences of problems, like manual code edits by the developer outside of the live-tracking system. While there are various aspects to improve on, in particular concerning the restriction of manual code edits, the system in its early form showed great promise for a deep understanding between running application and code.

6.1 Introduction

A well-built user interface (UI) not only functions well, but is also aesthetically pleasing and easy to comprehend for users [27, p.26]. As such, UI design is a domain that requires developers to work on visual aspects of their application. To help with the visual work in UI design, many approaches have been developed. For one, designated UI design tools present developers with a skeleton of their application. They can tweak the visual aspects of this skeleton to their liking and later connect it to the actual application code. Another approach is live programming. Live programming aims to make changes in the code more tangible by continuously presenting the output of the program or individual steps of it. On the other hand, in traditional compiled languages and without UI designer tools, developers have to imagine every change they want to apply to the UI, apply it to code, compile, and run their application. Only then, they can verify their intuition on what the change should look like. Next to tools that help building the UI, there are also tools that allow editing the visuals and state of the running application via inspectors or other methods. Upon determining the desired changes by live-editing the application, developers can apply the changes in the code without having to rely solely on their intuition.

The above approaches however all require maintaining two versions of the application: one in the developer's imagination as they simulate the code responsible for the UI or imagine how the application will look outside their UI designer tool and another as they actually run it and state and dynamic behavior come into action. Live programming gets closest to alleviating this problem, by presenting developers with an accurate representation of the consequences that their changes in code have. But live programming is usually limited to a very narrow domain and as such developers have to reinvent most tool support for it, because making sure that the application can be continuously executed requires careful consideration about its state [4, 8, 26].

Ideally, developers would be able to reuse their existing tool support for live-editing user interfaces or other domain specific applications, while not having to reenact the changes in the code to actually save the live-edit changes for future executions of the program. In this work, we will demonstrate an approach that enables existing tool support for live-editing running applications to modify the application's source code to persist the live-edit changes automatically.

With our *live-tracking system* we present an approach that allows developers to interact with the running application, navigate between pages and trigger states changes in any way they wish. Should they note an area where a visual improvement is necessary, it can be applied without the need to use the UI designing tool or go back to the code base. Instead, they can apply a change using their live-editing tools, directly on the interface they are seeing. Our live-tracking system is notified about this change, automatically finds a well-suited place in the application's source code and persists the change. Shneiderman demonstrated in his work about direct manipulation that having a short feedback cycle can foster creativity and experimentation and reduce frustration with tools [33, pp.192]. As this approach allows working directly with the objects themselves, we hope to bring the possibilities of direct manipulation to any domain that supports live-editing tools. The implementation we will describe later is designed to be independent of specific frameworks and can be easily integrated with existing frameworks from any domain, not just user interface design.

In section 6.2, we will explain the relevance of live-editing and introduce some of the core concepts our live-tracking system needs. In section 6.3 we will outline our approach and what challenges this concept poses. We will then describe further problems that arose during implementation and how we solved those in section 6.4 and discuss the live-tracking system's concept from a technical and a user-experience point of view in section 6.5. Lastly, we will provide an overview of related work in section 6.6, consider aspects for future investigation in section 6.7 and evaluate the impact of our live-tracking system in section 6.8.

6.2 Background

In this section, we will take a closer look at what our system achieves, then introduce some concepts needed for our live-tracking system and outline a concrete example on how it will improve the workflow of a user interface designer or programmer.

6.2.1 Feedback Cycle

Our goal is to minimize the time it takes a developer after modifying the user interface to seeing it live, that is in the actual application. We will call this minimizing the feedback cycle. As a consequence, we aim to also minimize the number of tools a developer has to interact with to run once through the feedback cycle. We will talk of context switching each time a developer has to change to a different tool on their way to seeing the result of their work.

Clearly, an approach like direct manipulation is ideal for our goal of minimizing context switching and the feedback cycle, since the developer is seeing the result of their program in front of them while applying the needed changes. The modifications are applied immediately through special interactions or overlaid handles like a *halo*, which we will describe in subsection 6.2.2, without requiring the developer to go back to their editor. In contrast, the typical way to develop with compiled languages like C/C++ requires the developer to go to their editor after observing the need for a change, find the piece of code responsible for the required change, adapt it, switch to their compiler, possibly also their linker, run the program, navigate back to the place in the program where they originally observed the problem and finally verify if the change had the desired impact. In other words, a high number of context switches and a long feedback cycle.

Approaches to reducing this feedback cycle include concepts like the REPL — read-evaluate-print-loop. This interface, however, is inherently not visual and thus harder to experiment with. It requires copying the final code over to the source code once satisfied with the results. Live environments like most Smalltalk implementations allow the programmer to change code while their application is running, taking immediate effect. For user interfaces, however, most of the code responsible for the visuals of the application is located in initialize methods that will only be run once on application start-up, rendering the live-editing useless for our particular case. Dedicated user interface design applications let developers work on an isolated, static copy of their user interface without behavior and thus cannot display complex state changes that may be an integral part of a user interface.

6.2.2 Modifying Living Objects

As described by Booch, each object is defined by its identity, behavior and state [3]. In the Squeak/Smalltalk environment [17], objects communicate by sending messages, which is also the only way to modify the state of the object from an outsider's point of view on the object.

Modifying state is a well-supported operation in a live environment such as Squeak/Smalltalk. In Squeak/Smalltalk, user interfaces use the Morphic framework by default [25]. A user can modify any user interface (UI) element of the Morphic framework in two ways. First, a *halo* that displays these modification options can be invoked around any element by option-clicking it. See Figure 6.1 for an example. This halo has a variety of handles, the colorful round buttons. These allow different actions to be applied on the morph in the spirit of direct manipulation, for example resizing, rotating, or changing its color. All interactions with the halo cause messages to be sent to the halo-ed object, in order to modify its internal state, for example position or extent.

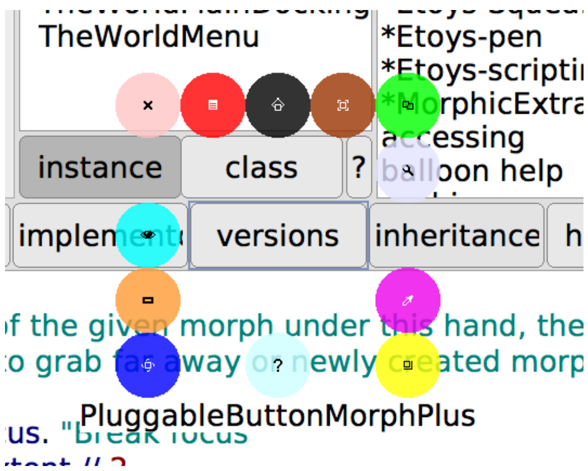


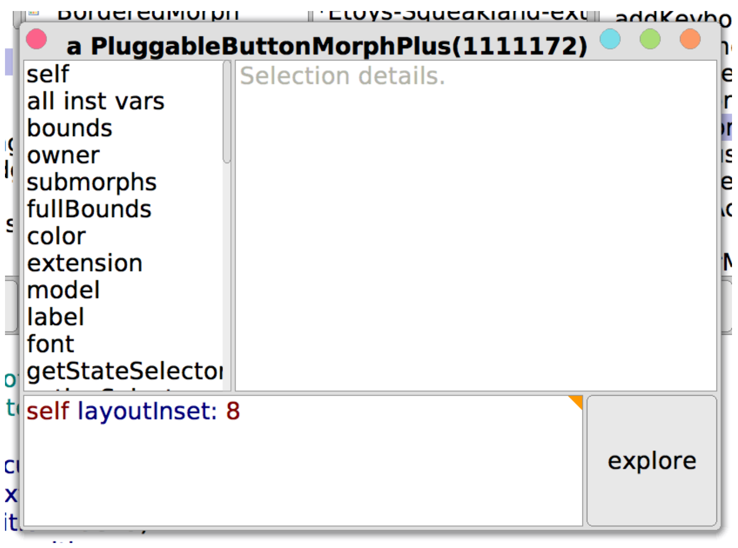
Figure 6.1: The user can invoke a *halo* around a morph from the Squeak user interface framework Morphic by option-clicking on the morph.

Additionally, the gray wrench icon will open an object inspector as seen in Figure 6.2. The inspector displays the state of the object by listing all of its instance variables and their current values. It also contains a code input field that can live-evaluate any entered code against the inspected object. This way a user can easily invoke a setter that changes a visual property. They could for example invoke the halo around the button, as seen in Figure 6.1, open the object inspector and evaluate a statement like `self layoutInset: 8` to change the inner spacing of the button, an operation that is not found in the direct manipulation handles of the halo itself. In this moment, they are sending the `#layoutInset:` message to `self`, which is the inspected object, with the argument `8`.

Of course, these operations have the downside that they are not persisted. Even though a user can tweak a program to their liking by modifying any state they want, as soon as they close the program their changes will be lost. Upon opening the program again, they would have to start over.

6.2.3 An Example

Our working example for this work will be GlyphHub, which is a review tool for font designers that eases the feedback process for fonts and makes sharing easy by storing all feedback on a central server. GlyphHub contains a variety of pages that a user can navigate to. In this example we look at the *sample text overview page* seen in Figure 6.3, a list of texts in different writing systems for font designers to test their fonts.


a PluggableButtonMorphPlus(1111172)

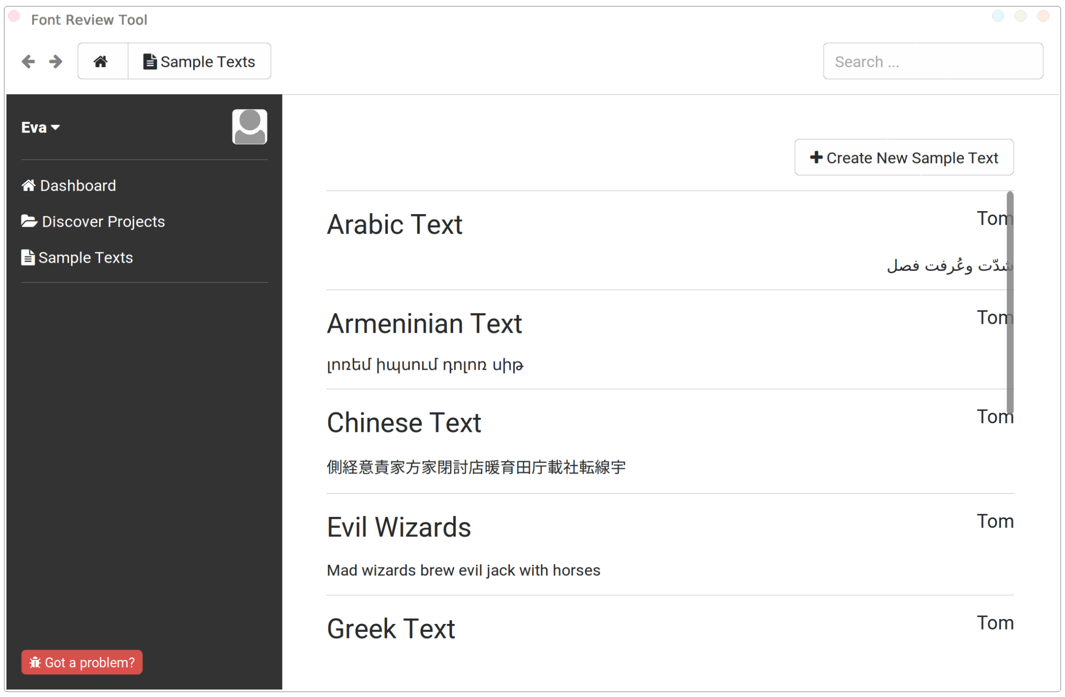
self
all inst vars
bounds
owner
submorphs
fullBounds
color
extension
model
label
font
getStateSelector

Selection details.

```
self layoutInset: 8
```

explore

Figure 6.2: The object inspector in Squeak. It displays properties of the inspected object and allows evaluating code with the inspected object bound as `self`.


Font Review Tool

Sample Texts

Search ...

Eva

Dashboard
Discover Projects
Sample Texts

Got a problem?

Create New Sample Text

Arabic Text — Tom
شدّة وعُرفت فصل

Armeninian Text — Tom
լոռեմ իպսում դոլոռ սիթ

Chinese Text — Tom
側経意責家方家閉討店暖育田庁載社転線宇

Evil Wizards — Tom
Mad wizards brew evil jack with horses

Greek Text — Tom

Figure 6.3: The sample text overview page of GlyphHub. Note the missing spacing between the scrollbar and the content.

During testing, a developer might have navigated to that page after logging in to the system with their account. They note that text on the right is colliding with the scrollbar. In an environment without live editing tool support, the following steps will likely be necessary to fix this visual problem:

1. Guess the class name or search for the class that builds the sample text page.
2. Open it in the programming environment and locate the part that builds the scroll area.
3. Make a guess on a suitable value for additional padding on the right or look up the width of the scrollbar in framework code.
4. Compile and run the new code.
5. Navigate to the same page.
6. Verify that the chosen value looks appropriate and if not, repeat the process from step 3.

With live-edit tool support, however, the process may be simplified to this scenario:

1. Inspect the scroll area on-screen with the live-edit tool.
2. Change the padding value on the right until it looks good.
3. Locate the class that builds the sample text page, often the class name will be displayed while inspecting the object.
4. Locate the code that builds the scroll area and insert the previously tested value.
5. Compile if necessary and optionally run the application to check.

If the first guess on the value was right in the first example, both approaches require the same number of context switches. For the likely case that the value needs to be tweaked at least twice, however, the first approach will involve more switching, since it requires editing values in the programming environment and inspecting the result in the application. Live-editing tools simplify this by providing editing capabilities right inside the running application.

We do note, that the programming environment with the live-editing tool hypothetically already has all the needed information to understand the developer's intent: change the padding of the scroll area to a certain value. It still requires manually switching to the development environment, locating the source code and manually changing the value again, this time in code, to make it persistent.

The ideal case could be for the live system to know, after the developer changed the living object, that the new value may be persisted. Or even that the live system persists the changes the developer makes simultaneously in the code. We would thereby drop the necessity to remember values that were found to be fitting, until the developer finds the piece of code where this change is to be made. Especially, if they are tweaking multiple properties that need to harmonize with each other, like the padding on the left and right and spacing between each item, they will have to remember each of the corresponding values and hunt down each place to persist them. Dropping the need for manually persisting allows the developer to stay within the user interface context for a lot longer.

6.3 A Live-edit Tracking System

In this section we will describe the approach for a live-tracking system that can detect changes by the developer on the living objects and automatically write them into code. In order to do so, the system needs to be able to (1) find out about all relevant state changes and (2) produce new syntactically correct methods that incorporate those changes. Before that, we need to define what data is relevant to this system by defining two types of events our live-tracking system uses.

6.3.1 Types of Events in a Live-tracking System

In order to be notified of state changes, we track message sends in the environment and generate events when our trackers are triggered. These events, depending on the context of the sender, may reflect different types of events. Either a *manual* change of the developer to the running program, such as an edit via the halo. Or a *planned* state change as written code in the program's own source code, such as setting the color of a morph to red inside an initialization method. We will refer to the former as *live-edit events* and the latter as *application events*.

6.3.2 Structure of The Events

Both live-edit and application events are associated with a specific set of data when they occur. For application events, the data is required to find the place in the source code where the method call was invoked from, and thus the place where we would need to change it later. Application Events are tuples consisting of these fields:

Instance, Selector A reference to the tracked instance on which the method that generated the event was invoked and the method's selector.

Arguments The list of argument values passed to the invoked method.

Source Instance, Selector The instance and selector from which the tracked method was invoked. In other words, usually the owner of the method just above the tracked method in the call stack. The source instance and selector identify where we change source code.

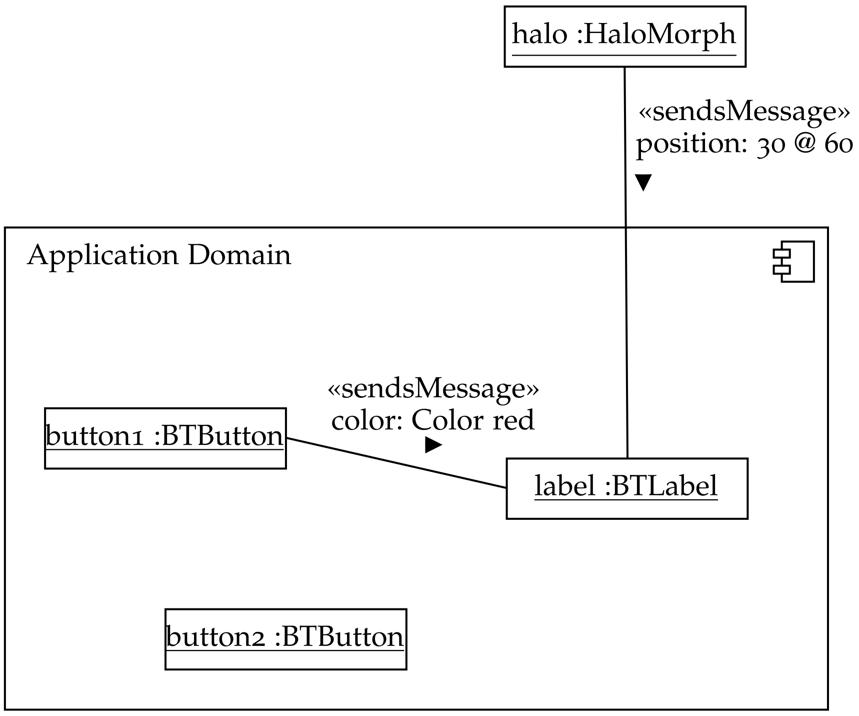


Figure 6.4: UML object diagram that illustrates a set of user interface objects belonging to our application. We distinguish between two types of events. *Application events*, generated from messages sent from objects inside the application domain to other objects inside the domain. *Live-edit events*, generated from messages sent from objects outside the application domain to objects inside the domain.

Source Interval The string indices into the method source code of the above method where the tracked method was invoked.

Source Timestamp The timestamp of the last edit on source instance's source selector method, which we will use to resolve conflicts as described later.

Live-edit events have the first three fields with application events in common. They add one optional field:

Instance, selector, arguments Same as above.

Source String The exact string the developer evaluated, if it can be extracted from the context. It can be used to express more complex expressions, rather than just using the final argument values.

To find matching application events for a given live-edit event, one can simply check for identity equality on the first two fields of both events, the instance and selector.

6.3.3 Tracking Events

For tracking the events, we use *method wrappers*, which allow replacing the method of a class with code that is to be evaluated instead. Using this concept, we can define code that builds one of our two event types and emits it accordingly whenever one of our wrapped methods is called. We then proceed to invoke the wrapped method and let the application continue as normal.

Once our wrapped method was triggered, we can inspect the stack that lead to the call. By letting creators of tools for live-editing inform the tracking system of a *stack frame matcher* that identifies stack frames belonging to their tool, we can now distinguish whether we are dealing with a live-edit or application event. We simply walk up the stack and check whether any of the live-editing tool matchers apply, in which case we found a live-edit event.

For example, when a developer changes the alignment of a button via the object inspector seen in Figure 6.2, the following things happen behind the scenes: the developer opens the object inspector on the button and changes its alignment in the layout to make it appear right instead of left. When they execute the `self align: #right` statement on the button, our method wrapper around the `#align` method gets invoked. Looking up the stack, we find that there are a number of method calls before the alignment was actually changed, dealing with evaluation, parsing, and eventually handling the keyboard input for the *execute* shortcut. We will also find a particular `#evaluateSelectionAndDo:` method on the stack, which is the entry point for the live evaluation in the object inspector right after the keyboard handling. By matching against this method and the class that received the message, we can reliably identify that this change of alignment was in fact triggered by a live-edit event.

Stack frame matchers for the object inspector, which is simply the evaluate method of the default text editor in Squeak, and Squeak's halo can look like this:

Listing 6.1: Matcher to identify stack frames belonging to the object inspector and the halo, which is described in subsection 6.2.2

```
"all stack frames of messages to a HaloMorph instance"
BTLStackMatcher forClass: #HaloMorph

"all stack frames of #evaluateSelectionAndDo: sent to a SmalltalkEditor instance
The object inspector includes a SmalltalkEditor. The #evaluateSelectionAndDo:
method is invoked just before the source code string is handed to the compiler"
BTLStackMatcher forClass: #SmalltalkEditor selector: #evaluateSelectionAndDo:
```

6.3.4 Locating Method Calls in Application Source Code

The generated events will form a stream of live-edit and application events. Upon opening the application, many application events will likely be generated, caused by the initialization code of the user interface that sets up state like colors or alignment of buttons and other widgets. During this phase, we learn the most about the structure of the application and where the existing code that is relevant to us is located.

For application events, we also need the location of the method invocation, the source receiver, selector and interval in our application event tuple. This location can be determined in Squeak/Smalltalk via the `debuggerMap` of any method. It allows mapping a program counter to a specific interval in the source string of the method. Once our method wrapper gets triggered, we query the method right above the tracked method on the stack for its debugger map and process counter. Using these, we calculate the source string interval and save it to the application event. Additionally, as described above, we of course also remember from which method selector and class we saved the indices. This way, the application events in the event stream form a history of properties that were modified on the widgets of our application and where these modifications came from.

As soon as the live-tracking system is informed of a live-edit event, it inspects the event stream in reverse chronological order. The system then tries to find an application event that matches the live-edit event in terms of the selector of the used method and was sent to the same instance. For the live-edit event from Figure 6.5 this would be `#color:` and `a BTButton(419293)`. Once it found such an event, the system can use the saved source code location to change the argument of the call to the new value. In the example that would be `#initialize` of the `CommentPage` class, between indices 12 to 32 in the source code.

Walking the stream in reverse chronological order has the effect that, if there are multiple application events matching our corresponding live-edit event, we always change the code that was executed after the most recent change of state in the application. Considering a new example where a status display turned red in our application, after it was initialized to green, and we would change its color while its red, we will change the place in the code that set it to red, instead of the initialization code that set it to green. We can assume that this is almost always exactly what the developer wants to happen, that is changing the visuals corresponding to the exact state of the application they are in right now. In any

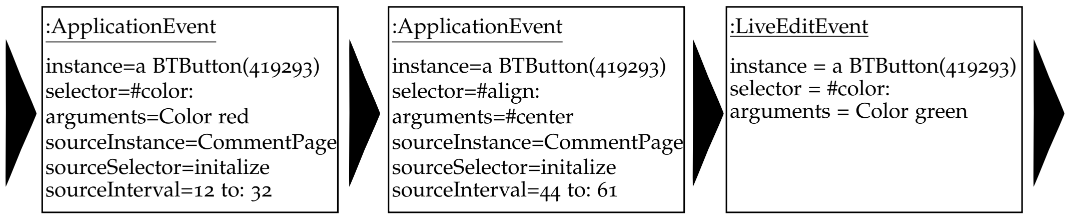


Figure 6.5: Excerpt from an event stream produced by opening an application containing a button of which the color was set to red and its alignment to `#center`. After that, the developer changed the button's color to green via a live-editing tool.

other case, it would be an option to display the developer with the history of the property to pick from which we can easily obtain from the event stream.

6.3.4.1 Application Instantiation Events

In the case that we cannot find a matching application event for a given live-edit event, which means the application code has never set this property before, we are faced with a considerably more difficult situation. To apply this live-edit event without a matching application event, we need to insert entirely new code, instead of simply changing the argument of a single invocation at a place that is well known to us thanks to the information we save in each application event.

To deal with this situation, we also track the initialize methods of classes we are interested in. Whenever an initialize method is called, instead of going just one stack frame up to find the sender, we go up until we "leave the class". For example, if we have a class-side construction method `newWithLabel:` for the `BTButton` class, this method will likely first create an instance of its class via `Behavior class>>new`, which in turns calls our initialize method. Since we watch the initialize method, our method wrapper is now invoked. We then walk up the stack until the receiver of the call is no longer the object itself, a `BTButton`, or the object's class, the `BTButton class`. This way we will find the method that called the `newWithLabel:` method. We now emit an application event that will carry the special semantic of being the instantiation place. We will thus refer to it as an *application instantiation event*. The described example can be seen in Listing 6.2.

Listing 6.2: Example for finding the place where a tracked object has been instantiated by "outside" code, that is code not belonging to the class of the instantiated object.

```
MyWindow>>initialize
  self add: (BTButton newWithLabel: 'Hello!') "this is the place we want to find"
  ...

BTButton class>>newWithLabel: aString
  ↑ self new label: aString; yourself

BTButton class>>new "actually defined on its Behavior superclass"
  ↑ self basicNew initialize
```

```

BTButton>>initialize "method that we wrap"
  ...
```

In the example above we would add another pair of parentheses around the creation of the BTButton in `MyWindow>>initialize` and then add a new method call like `color: Color red` inside those:

```
MyWindow>>initialize
  self add: ((BTButton newWithLabel: 'Hello!')
    color: Color red;
    yourself).
  ...
```

Now, with these basic building blocks for our system, we are able to comprehend how our developer's application is built, record whenever they trigger a live-edit and apply their change to the source code of the application at the right place. On a conceptual level, this allows developers to adjust any property of their running application without the need to switch to a programming environment.

6.4 Implementation

In this section, we will examine challenges we encountered during implementation and how those can be resolved. Namely, we will show how the tracking can be implemented via annotations on methods (subsection 6.4.1) and how the event stream comes to be (subsection 6.4.2). In addition to that, we demonstrate how to "shield" our live-tracking system against changes by developers outside of our system (subsection 6.4.3) and examine two aspects of modifying the source code of existing methods in practice (subsection 6.4.4 and subsection 6.4.5). Lastly, we illustrate how dedicated tool support for this live-tracking system can be created and what an existing live-editing tool would need to do to support our system (subsection 6.4.6). Our proof-of-concept implementation is publicly available on GitHub[1].

6.4.1 Annotating Methods

In order to note all relevant changes, we would hypothetically need to listen to every single message send in the system. Since we are only interested in methods that change the visual state of an object, we introduced a way of marking relevant methods with an annotation, also called *pragma* in the Squeak/Smalltalk environment.

Replacing all methods in the environment is not an option, since the overhead necessary to store events by itself also requires message sends. So, while trying to

[1]Proof-of-concept implementation of the live-tracking system `https://github.com/tom95/BTLiveEdit`.

Listing 6.3: An annotated method that changes the color of a morph

```
Morph>>color: aColor
  <liveEditable>

  color := aColor.
  self changed.
```

note the occurrence of an event, we would trigger new events that we want to note, bringing us into an infinite loop.

We then offer methods to begin a live-edit session via a `BTLiveEditSession` on all methods marked with `<liveEditable>`, or only on the marked methods in specific system categories. This could for example only be the categories from a developer's application and the categories from the frameworks they use.

While implementing this feature in the setters of a framework, it is likely that a distinct pattern for which methods are interesting will be noticed. Filtering automatically by assignment to a single instance variable that has the same name as the method will probably produce the desired result without requiring to manually add annotations. Annotations, however, also provide an opportunity for tools as we will show later in this section.

6.4.2 Organizing Live-edits in a Session

In our implementation, we decided to wrap the live-tracking system in a session. Instead of listening on message sends at all times, we only do so when the developer explicitly invokes the live-tracking system when starting their application. It would have also been possible to have the system running all the time. Old events will not cause trouble, since we verify that the timestamp of the event occurrence equals the method's current edit timestamp before doing anything with the method, as seen in subsection 6.4.3. However, the list of events would quickly grow, especially if the programming environment itself is built with a live-tracked framework and keeps emitting events. Whether or not the session is active all the time has some interesting consequences.

Clearly, not tracking messages all the time will improve performance, since we will not have the overhead for emitting and handling the application events. This way, developers can enable the session when they are working on the user interface or debugging and disable it again when shipping their application to customers or testing performance critical aspects of their application.

In particular in live environments like Squeak/Smalltalk that support fix-and-continue, there is great interest in debugging and improving applications that are alive and running. A major downside of having the live-tracking limited to a session is thus that activating the session on a currently running application may produce incorrect results. Since we only tune in later, we were not tracking large parts of the application's history and are in particular missing all calls from the initialize methods in the event stream. With this limited knowledge, the live-tracking system would constantly assume that no matching application event exists for given live-edit events and thus might accidentally insert duplicate calls.

To support developers, we provide a convenience mechanism to enter and leave the mode automatically. Upon running `MyApplication new openTracked`, the live-tracking system activates and subscribes to the `deleted` event of the widget. Once the `deleted` event is emitted, which our widget morph subclass does whenever it is closed, we stop the live-tracking system.

6.4.3 Dealing With Manual Code Changes

Changes to the source code outside of our live-tracking system will generally invalidate all application events in the event stream for this particular method that occurred thus far. Consider this example:

1. The code constructs a widget and sets its color to red. We record an application event for the range of for example `12 to: 24`.
2. If the developer now changes the color to green via a live-edit tool, we change the widget class's `#initialize` method by substituting from the range 12 to 24 in the string with the new value.
3. Next, they also manually add a call to change our widget's spacing to 30, right above the call to change the color. The indices for the color call have now increased as new code has been inserted above the call.
4. Applying the next live-edit to color would now write right over the new spacing call, possibly resulting in invalid syntax, but in any case not doing what the developer intended.

We can prevent erroneous edits from our live-tracking system from happening by remembering the last edit timestamp of the method at the time when we determined the indices. The edit timestamps are updated by the environment whenever a method is modified. If this timestamp changed by the time we want to apply a live-edit event, we cancel the operation or prompt the developer to apply the operation manually by presenting an editor with the according method. A more sophisticated approach could attempt to comprehend the changes the developer applied to the function and try to figure out what the new indices for our from-source event are.

Since any change of our live-tracking system also updates the timestamps, applying any live-edit event would invalidate all application events for that method. We can however benefit from the fact that we exactly know how the live-edit event changes the code, since it is the live-tracking system's responsibility to apply it. We can thus iterate over all affected application events, update their source intervals according to the applied operation and then bump their timestamps.

6.4.4 Reconstructing Live-evaluated Code

Code that is evaluated via the object inspector is one example where we can enrich the resulting live-edit event with more information, since the developer can describe more complex assignments, such as passing an instance variable to the live-tracked method. By traversing the call stack up to the point where we find the original `CompilationCue` from the object inspector, we can get hold of the exact string that the developer evaluated. Parsing this expression and extracting only the part that forms the actual message send just after the receiver, we end up with the portion of code that we will need to replace the previous code piece with, possibly containing variables. For example, a live-evaluated code piece like `self color: self owner color` would be truncated to just `color: self owner color`, so that it will replace exactly the range of the method invocation.

We can, however, not always be certain that the variables will be resolvable in the context where we will insert them. A quick test-compile of the method will reveal such a case. If that happens, we fall back to generating the necessary code for the arguments ourselves by calling the `storeOn:` method on each argument, which instructs the object to write a representation in code of itself on a given stream. Using this representation and the name of the selector we save in the live-edit event, we can build the same string as above, just without variables.

Listing 6.4: An OrderedCollection containing the numbers 2 and 3 may turn into this string representation. Evaluating this string of code again will result in a list object with the same contents.

```
'((OrderedCollection new) add: 1; add: 2; yourself)'
```

Since the usage of `self` always depends on the context, we could verify that the expression is valid in the new context by also extracting the receiver from the `CompilationCue` and checking whether the object that triggered the application event is the same as the `CompilationCue`'s receiver. If that is not the case, the message was not sent directly to the instance and we would not do as intended if we were to update the method with this code, as shown in the example in Listing 6.5.

Listing 6.5: The message `color:` is sent to a different instance than `self` in this expression. Just replacing the original call with `color: self color` would thus be incorrect, even though all variables can be resolved.

```
self owner color: self color
```

This way, we can offer the developer an opportunity to express more complex relationships between data than just simple assignment of constants, while still making sure that we will always end up with a valid method. Again, one may consider prompting the developer with a choice if a variable he used was not resolvable.

6.4.5 Inserting New Method Calls

As described in subsection 6.3.4, method calls to setters that have not been called before, already pose a challenge when trying to locate them. Namely, locating the place where the object is instantiated by code written by the developer and not code belonging to the framework. Once we found a suitable location for inserting the new setter, we still face the problem on how to insert the setter while doing as little changes to the existing code style in the method as possible.

In particular, nested construction calls that are typical for UI building are problematic here. Just producing a syntactically correct method is already complicated, even more so adapting to the developer's preferred code style as seen in the examples in Listing 6.6 and Listing 6.7.

Listing 6.6: Trying to set the color of the BTButton instance will result in a syntax error, unless parentheses are placed.

```
CommentPage>>initialize
  self add: BTButton new.
```

Listing 6.7: When setting the color of the BTButton, we need to be aware of the surrounding cascade and insert our call before the `#yourself`.

```
CommentPage>>initialize
  self add: (BTButton new
    align: #left;
    yourself).
```

We solve this problem by first building the abstract syntax tree (AST) of the method. We then locate the AST node that our location detection code identified as the initialize method, in the examples that would be `BTButton new`, and then analyze the immediate surroundings of this AST node.

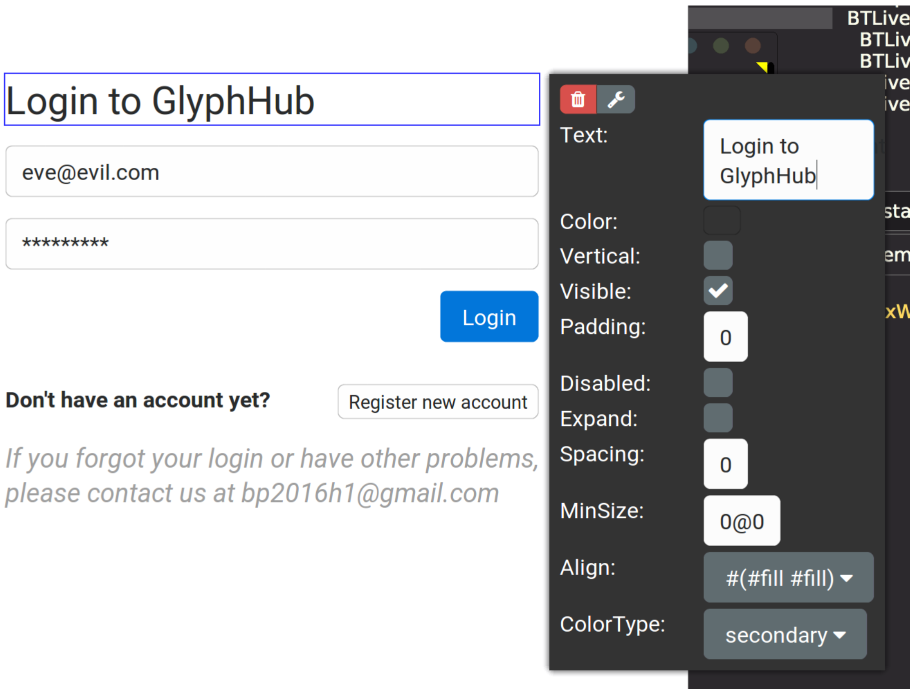
If we are in a cascade, we can simply add another call. If there already is a call, we need to start a cascade and append `yourself` if the value of the entire expression is used again. If there have been no calls so far, we will likely need to add parentheses and a cascade with `yourself`. If the developer is using a pretty-printer provided by the system, we may be able to simplify this to just inserting a new node in the AST and then telling it to auto-format into a string representation.

For inserting these method calls, there is a lot of potential for sophistication to make sure we adapt to the style the developer likes to code in, by for example analyzing other methods in the same package on their usage of cascades and line length limits. The approach by itself, however, is rather simple and comes to down to having considered all necessary corner cases.

6.4.6 Tool Support for the Live-tracking System

As outlined in subsection 6.4.1, integrating our live-tracking system in frameworks, or even application specific code, is simple. It suffices to mark which methods are to be tracked, whether that is via annotations or an auto-detect mechanism that analyzes methods. Of course, using the system only makes sense if tool support exists that allows modifying the state of the living objects. Here, we want to propose an addition to the annotation system to make it worthwhile employing annotations instead of using an auto-detect mechanism to considerably improve the tool support for live-editing.

So far, we have assumed that a developer somehow knows what live-editable properties their framework supports. This assumption makes our system more complicated to experiment with and requires remembering lists of properties, clashing with our goals of enabling developers to quickly try out settings without the hassle of finding them in the code.


Login to GlyphHub

eve@evil.com

****

Login

Don't have an account yet? Register new account

If you forgot your login or have other problems, please contact us at bp2016h1@gmail.com

Text: Login to GlyphHub
Color:
Vertical: ☐
Visible: ✓
Padding: 0
Disabled: ☐
Expand: ☐
Spacing: 0
MinSize: 0@0
Align: #(#fill #fill)
ColorType: secondary

Figure 6.6: Our property sheet, displaying buttons for deletion and inspection on top and the various property modifiers below

To make editable properties discoverable, we extended the `liveEditable` annotation to supply a class that defines a widget that can be used to modify the property in question by essentially providing extended type information. Examples for this extended notation can be seen in Listing 6.8. By overriding the default halo that opens on a morph, which we described in subsection 6.2.2, we can now collect all `liveEditable` annotations of the halo-ed class and its superclasses, build the modifier widgets for each property and then display them on-screen, right next to the selected morph. A possible implementation displaying dedicated editor widgets for numbers, points, boolean, enums and colors can be seen in Figure 6.6.

Listing 6.8: Possible annotation definitions for building a dedicated interface for adjusting property values

```
checked: aBoolean
  <liveEditable: #MyBooleanModifier>
  ...

align: aSymbol
  <liveEditable: #MyEnumModifier with: #(left right center)>
  ...

height: aNumber
  <liveEditable: #MyNumberModifier from: -10 to: 10 step: 0.1}>
  ...
```

Developers can supply their own modifiers by subclassing `BTLPropertyModifier`, `BTLPropertyModifierWithArgument`, or `BTLPropertyModifierWithInterval` and then adding the corresponding annotations as seen in Listing 6.8 to their methods.

This additional enhancement of the annotations is particularly useful if there are so many properties that building dedicated direct manipulation tooling like the halo will not be feasible anymore. However, if there are many properties, this property sheet enables developers to still maintain a good overview and easy ways of manipulating the connected data, rather than evaluating code manually. Especially user interface elements like sliders for changing numeric values open opportunities for experimentation that repeatedly evaluating the same method call with slightly different numbers cannot possibly match.

6.5 Discussion

In this section, we will discuss two technical issues related to the live-tracking system, take a look at opportunities for direct manipulation support and finish by discussing our own experience with using this system.

6.5.1 Loops and Dynamic Elements

Dynamic user interface elements, like widgets that are instantiated in a loop, have some interesting implications for our live-tracking system. Since we add the changed code right after the instantiation of the widgets, all widgets in the loop will always be affected by the change, which likely is exactly what the developer is trying to express. It does however mean that applying a change to just the first widget of a loop cannot be realized with this system, since this would require more advanced restructuring of the code, like adding a conditional in the loop or moving the instantiation of the first item to a place outside the loop.

Since live-editing tools typically only change properties of the one selected object, as the tools typically do not know the origin of an arbitrary object, it is up to the tracking system to provide the developer with a more coherent preview of his changes to widgets that are instantiated in a loop. To make the impact of the change more clear, we can extend the live-tracking system's matching behavior for mapping live-edit events to application events. Once we found one matching application event, we can check whether we also have application instantiation events, which are described in subsubsection 6.3.4.1, that have the exact same class, selector and source code location. If we find more matches than just the one from the originally edited instance, we can also apply the change to all other matching objects. This way, the developer will instantly see a change of color propagate to all widgets that were instantiated in the same loop, even though they only edited the first widget.

6.5.2 Ambiguities When Applying Changes

A challenge for both the live tracking system and the way developers have to think about the live-tracking system is how to resolve the ambiguity of where in the code to apply the live-edit events. Given any object, there are always at least two possible places: the object's initialize method, thus applying the change to all instances of the class, or right after its instantiation, thus applying the change to only this specific instance or rather all instances created by this specific piece of code.

The live-tracking system assumes in most cases that the developer means to modify only the instance he is seeing right now, since most of the classes with live-editable properties in our examples were part of the user interface framework code and would thus alter the framework's code if applied in the initialize method. In general, however, presenting the developer with a choice of *apply to all* versus *apply to only me* could be desirable for when the developer defines their own live-editable properties in their application domain.

A more advanced approach could be to mark certain classes or categories as immutable for the live-tracking system. This may increase the maintenance effort but could also be very straightforward for most cases, in particular for the typical case that most or all live-editable properties are declared in a framework. An automatic approach could limit applying live-edit events to the same category prefix that the opened application belongs to. For example, if there are GlyphHub-Core, GlyphHub-Settings and GlyphHub-Widgets packages and we use widgets from the ToolBuilder-Morphic package, all live-edit events can automatically be limited to classes that belong to a package with a "GlyphHub-" prefix, since the `GlyphHubApplication` main window that the session is started on belongs to the GlyphHub-Widgets package. See subsection 6.4.2 on how the session can connect to widgets.

Another problem with ambiguity arises from methods that do not directly map to state, but call other setters. Squeak's `Morph` class, for example, has separate setters for `bounds:`, `position:`, and `extent:`. The `bounds:` setter in turn only calls the `position:` and `extent:` setters. Suppose we record a live-edit event triggered by `bounds:` for which we find no matching application event. As there is no known place where `bounds:` has been set before, we will insert the new `bounds:` call after the instantiation of the morph. If later in the same method the developer calls the `position:` setter for any reason, our inserted `bounds:` call will suddenly be incorrect as the later call to `position:` overwrites parts of it. This way, the live-tracking system leaves the developer to believe that his change was applied successfully, but on the next run of the application, the conflict will become apparent as the position did not update.

If we do not track `bounds:`, but `position:`, our live-tracking system could be led to believe that it should update the framework's implementation of `bounds:`, since any previous call to the `bounds:` setter will have caused application events for the `position:` setter. To prevent this in frameworks that make use of convenience setters, the framework developer will need to make sure that all necessary methods are annotated correctly. Additionally, introducing immutability as described above would prevent the latter case.

6.5.3 Support for Direct Manipulation

Our event-tracking system by itself does not deliver any aspects of direct manipulation. It can however provide tools with opportunities to enrich their direct manipulation support with little effort.

Direct manipulation often is a way of experimenting, as "users can immediately see if their actions are furthering their goals, and if not, they can simply change the direction of their activity" [34]. One of the core principles of direct manipulation is offering easy ways to reverse any action. For an interface like the object inspector, it is however not immediately obvious how an action is reversible. Our live-tracking system can support tools in this way to support arbitrarily deep undo stacks. By filling a second stream of all applied live-edit events and the value they overwrote, we can provide tools with a complete edit history. If multiple tools are involved in the editing process, one may consider tagging live-edit events based on the stack frame matcher that identified the live-edit event, so that tools can display an undo action in their own interface without the developer getting confused why various actions not related to this particular tool are reverted.

A completely different approach to this could be to embrace the session concept more strongly. Instead of applying all operations to the code instantly, the live-tracking system only collects them inside the session, but still displays them right away. Once the session is closed, it can present the developer with a dialog akin to a "git" interactive staging interface, where they can select the changes they would like to apply and ignore the rest. This would, however, have the downside that new instances of objects would not run the modified code unless we add more complex synchronization code similar to what we added in subsection 6.5.1, possibly leaving the developer in an inconsistent or confusing state of part modified and part unmodified visual state.

6.5.4 The Live-tracking System in Use

We were only able to try the running live-tracking system in a not feature complete stage for a short while. However, in particular while testing the live-tracking system during its own development, the need for good fail-safes like test-compiles and immutability markers became apparent. Due to the initial lack of good heuristics concerning the place where live-edit events are to be applied, we remained rather suspicious towards the system and usually preferred running it in read-only mode until the session quit and offered us options to apply events manually.

The options provided did prove very useful in the user-interface-heavy Glyph-Hub application since we no longer needed to navigate first to the code, then to the same place in our application after restarting. However, we often found the need to express more complex changes. Tool support we created for the live-tracking system itself already allowed us to jump to the piece of code that was responsible for instantiating a specific widget or even set a specific property. This way, we found ourselves repeatedly invalidating large parts of the application events, as explained in detail in subsection 6.4.3, since we then edited the methods.

6.6 Related Work

The area around live coding has seen much work, both recently and in the past. One example that shares similar ideas to our approach is TouchDevelop [4]. TouchDevelop introduces a new language with a specific set of enforced restrictions concerning mainly the immutability of state from the point of view of a designated render function. The separation between code that can modify state and code for rendering allows TouchDevelop to replay the render code at any point and thus easily allows two-way modifications on both the user interface itself and the code.

Another notable example of combining programmatic and direct manipulation is Sketch-n-Sketch, a system that defines a small programming language to generate SVG documents [8]. The user may apply various operations to its rendered representation and the changes are applied back to the code. The variety of supported actions demonstrates the increased flexibility of such a live system, if there is no state to pay attention to.

Chugh envisions similar concepts to the ones presented in this paper [7]. He describes what he thinks systems will need to provide in order to bridge the gap between programmatic and direct manipulation. The combination of which he calls *prodirect manipulation*. According to Chugh, three modes need to be supported in a system that allows for *prodirect manipulation*: manipulating the output of the program should synthesize updates to the program code, output examples should suffice to synthesize code snippets and a third mode where automatic synchronization between code and application is temporarily suspended to allow for large changes and afterwards picked up again as good as possible. In our work, we essentially describe an implementation that matches the first mode.

An interesting approach to live-editing user interfaces, or rather graphical content in general, is found in the Lively Kernel [19]. Through simple composition of components found in a *PartsBin*, a developer can build their user interface and add scripts to individual parts of it. The result can then later be saved in the same PartsBin as well. By persisting not only code, but also the current state of any component in the PartsBin, a synchronization of state changes made via live-editing tools and code is not even necessary.

The concept of method wrappers that we use for live-tracking was also used to implement Aspect Oriented Programming (AOP) in the Squeak/Smalltalk environment [16]. The AOP implementation Aspect/S as described by Hirschfeld is powerful enough to support our live-tracking system. It would have been possible to implement the tracking part of the system in Aspect/S as two `AsBeforeAfterAdvice`s to achieve the same functionality that our own method wrapper implementation supports.

6.7 Future Work

There are multiple questions related to our approach that are worth investigating. We will describe some of those in the following.

6.7.1 Modeless Design

To optimize memory, the event-stream could be replaced by a dictionary, instead of the entire history of all events that happened while the session was active. This dictionary would hold application events hashed by their instance identity and selector and map them to the actual events. All references to the instance would only be weak references. This way, we do not save application events that our core mechanism of replacing the most recent occurrence of any application event does not need by only ever remembering the most recent event and once the user interface is closed and all references to the tracked object are freed, it would also be removed from our tracking dictionary on the next clean up, since it only held a weak reference. We would lose the ability of looking at the history of a property, but would reach the minimum required memory for running the live-tracking system itself.

This could be a first step towards dropping the requirement for a dedicated live-edit mode as described in subsection 6.4.2 as it would lower the memory footprint of having the live-tracking system run in the background.

6.7.2 Towards Two-way Synchronization

For a proper bi-directional synchronization between living object and code there is still a long route to go. As a first step, one could investigate how to keep application events valid, even after a method has changed. A possible approach could be to employ AST diffing as described by Falleri et al. to track where changes to the method where made and adjust the saved indices accordingly, or always keep the AST around and remember the node that was responsible for a tracked message send [12].

During user testing we were continuously noticing opportunities that tracking of all executed code provides. Supporting a read-only bi-directional mapping between source code and on-screen widgets is easily possible with the data we store in the session. When activating the halo on a widget, an editor popup could open, presenting the code that was responsible for creating the given widget. Similarly, upon marking a piece of code, one could imagine the concerned widgets to get highlighted in the user interface next to the editor. Even though the previously described challenges coming from manual code changes block us from supporting a synchronization between the displayed source and the widgets, this feature could already be immensely helpful for reverse engineering a legacy user interface system.

6.7.3 Further Testing

Due to the time constraints while testing the system, we cannot make confident statements on the improvements for developers thanks to our our live-tracking systems. Interesting questions that are left to be evaluated include:

> Is forbidding manual edits to the code while live-editing with our system an option or will developers keep invalidating the event-stream because they feel that this limits their expressiveness too much?
>
> How advanced would the live-editing tools need to be to support building the entire user interface without touching code, starting from an empty application window? Is the live-tracking system already advanced enough to support this or will it require dedicated logic to support composition of widgets?
>
> How well will the live-edit mode work in performance critical applications like games? Will the lag be so detrimental for the experience that the benefits of increased live-editing capabilities are completely negated, since no real testing can be performed?

6.8 Conclusion

The live-tracking system we described allows developers to use any of their existing live-editing tools to make persistent changes to their running application. Our approach further opens opportunities to create a deep connection between the running application and the code that produced it. During our testing, we felt well-supported by the ability to quickly map from visible properties to pieces of code and the ability to try a variety of values without much context switching. As said in subsection 6.5.4, building trust towards a system that aims to make things happen behind the scene was hard. We think, however, that the opportunities and advantages of our approach will outweigh any uncertainties towards the live-tracking system, in particular as the heuristics for code placement and fail-safes improve. All in all, we regard the live-tracking system as a promising step towards having an agnostic opt-in system to provide developers with an experience that allows them to have a deep connection between code and living objects with immediate opportunities to modify anything they are seeing, persistently.

7 Conclusion

In this report we have described our work on a platform to aid the font creation process and to how Squeak/Smalltalk has been helpful for us to rapidly create prototypes and deeply integrate this domain specific system with the Squeak/Smalltalk host system.

We demonstrated and discussed techniques that helped us advance more quickly, such as rapid prototyping and code generating based on live-edits. We described the basic building blocks needed to build the system, such as complex text rendering and vector glyph editing, and how we approached their integration into the Squeak/Smalltalk system. The font review platform itself still needs to be tested on a larger scale. Early feedback from font designers indicated that many of the explored concepts could provide a massive increase in the speed of their creation process. We believe that the concepts and techniques we described for the creation of this prototype can provide a helpful starting point for people approaching a similar domain or continuing work on this project.

References

[1] C. Abras, D. Maloney-Krichmar, and J. Preece. "User-centered design". In: *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications* 37.4 (2004), pages 445–456.

[2] J. R. Behdad Esfahbod Nys Martensen. *HarfBuzz*. URL: `https://www.freedesktop.org/wiki/Software/HarfBuzz/` (visited on 2017-07-31).

[3] G. Booch. *Object-oriented Analysis and Design with Applications (2nd Ed.)* Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994. ISBN: 0-8053-5340-2.

[4] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. "It's Alive! Continuous Feedback in UI Programming". In: *SIGPLAN Not.* 48.6 (June 2013), pages 95–104. ISSN: 0362-1340. DOI: `10.1145/2499370.2462170`. URL: `http://doi.acm.org/10.1145/2499370.2462170`.

[5] M. Carr and J. Verner. "Prototyping and Software Development Approaches". In: (July 2017).

[6] G. Castelnuovo, A. Gaggioli, F. Mantovani, and G. Riva. "From Psychotherapy to e-Therapy: The Integration of Traditional Techniques and New Communication Tools in Clinical Settings". In: *CyberPsychology & Behavior* 6.4 (2003), pages 375–382. DOI: `10.1089/109493103322278754`. URL: `http://online.liebertpub.com/doi/pdf/10.1089/109493103322278754`.

[7] R. Chugh. "Prodirect Manipulation: Bidirectional Programming for the Masses". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE '16. Austin, Texas: ACM, 2016, pages 781–784. ISBN: 978-1-4503-4205-6. DOI: `10.1145/2889160.2889210`. URL: `http://doi.acm.org/10.1145/2889160.2889210`.

[8] R. Chugh, J. Albers, and M. Spradlin. "Program Synthesis for Direct Manipulation Interfaces". In: *CoRR* abs/1507.02988 (2015). URL: `http://arxiv.org/abs/1507.02988`.

[9] E. community. *Sueakland: home of Squeak Etoys*. URL: `http://www.squeakland.org/` (visited on 2017-07-31).

[10] C. Courage and K. Baxter. *Understanding your users*. Morgan Kaufmann Publishers, 2005.

[11] D. Crockford. "The application/json media type for javascript object notation (json)". In: (2006).

[12] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. "Fine-grained and accurate source code differencing". In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 2014, pages 313–324. DOI: 10.1145/2642937.2642982. URL: http://doi.acm.org/10.1145/2642937.2642982.

[13] F. Forssman and R. de Jong. *Detailtypografie: Nachschlagewerk für alle Fragen zu Schrift und Satz*. Herrman Schmidt, Mainz, 2004. ISBN: 978-3-87439-642-4.

[14] A. Goldberg and D. Robson. *Smalltalk-80 The Language and its Implementation*. Xerox Corporation, 1983. ISBN: 0-201-11371-6.

[15] J. Guertler and J. Meyer. *Design Thinking*. GABAL Verlag, 2013.

[16] R. Hirschfeld. "AspectS - Aspect-Oriented Programming with Squeak". In: *Objects, Components, Architectures, Services, and Applications for a Networked World: International Conference NetObjectDays, NODe 2002 Erfurt, Germany, October 7–10, 2002 Revised Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pages 216–232. ISBN: 978-3-540-36557-0. DOI: 10.1007/3-540-36557-5_17. URL: https://doi.org/10.1007/3-540-36557-5_17.

[17] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. "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*. OOPSLA '97. Atlanta, Georgia, USA: ACM, 1997, pages 318–326. ISBN: 0-89791-908-4. DOI: 10.1145/263698.263754. URL: http://doi.acm.org/10.1145/263698.263754.

[18] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself". In: *SIGPLAN Not.* 32.10 (Oct. 1997), pages 318–326. ISSN: 0362-1340. DOI: 10.1145/263700.263754. URL: http://doi.acm.org/10.1145/263700.263754.

[19] D. Ingalls, T. Felgentreff, R. Hirschfeld, R. Krahn, J. Lincke, M. Röder, A. Taivalsaari, and T. Mikkonen. "A World of Active Objects for Work and Play: The First Ten Years of Lively". In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2016. Amsterdam, Netherlands: ACM, 2016, pages 238–249. ISBN: 978-1-4503-4076-2. DOI: 10.1145/2986012.2986029. URL: http://doi.acm.org/10.1145/2986012.2986029.

[20] C. Jaschek. "Iterative Software Prototyping Through the Example of a Font Review Tool". In: *The Font Engineering Toolkit: Live Font Creation in a Self-supporting Programming Environment*. 2017.

[21] E. Krebs. "Viability of Complex Font Rendering in Live Environments". In: *The Font Engineering Toolkit: Live Font Creation in a Self-supporting Programming Environment*. 2017.

[22] A. Löser. "Live Glyph Editing in a Live Environment". In: *The Font Engineering Toolkit: Live Font Creation in a Self-supporting Programming Environment*. 2017.

[23] V. H. Mair. "Modern Chinese Writing". In: *The world's writing systems* (1996), pages 200–208.

[24] J. H. Maloney and R. B. Smith. "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. Pittsburgh, Pennsylvania, USA: ACM, 1995, pages 21–28. ISBN: 0-89791-709-X. DOI: 10.1145/215585.215636. URL: http://doi.acm.org/10.1145/215585.215636.

[25] J. H. Maloney and R. B. Smith. "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. Pittsburgh, Pennsylvania, USA: ACM, 1995, pages 21–28. ISBN: 0-89791-709-X. DOI: 10.1145/215585.215636. URL: http://doi.acm.org/10.1145/215585.215636.

[26] S. McDirmid. "Usable Live Programming". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: ACM, 2013, pages 53–62. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509585. URL: http://doi.acm.org/10.1145/2509578.2509585.

[27] J. Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 0125184050.

[28] J. Nielsen. *Usability engineering*. 1st edition. Kaufmann, 2009.

[29] D. A. Norman. *The Design of Everyday Things*. 2nd edition. New York, NY, USA: Basic Books, Inc., 2002. ISBN: 9780465067107.

[30] H. Plattner, C. Meinel, and U. Weinberg. *Design thinking*. mi-Wirtschaftsbuch, FinanzBuch, 2009.

[31] J. Preece, Y. Rogers, and H. Sharp. *Interaction design*. J. Wiley & Sons, 2002.

[32] A. N. Schulze. "User-Centered Design for Information Professionals". In: *Journal of Education for Library and Information Science* 42.2 (2001), page 116. DOI: 10.2307/40324024. URL: http://www.jstor.org/stable/pdf/40324024.pdf.

[33] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201694972.

[34] B. Shneiderman. "The Future of Interactive Systems and the Emergence of Direct Manipulation". In: *Proc. Of the NYU Symposium on User Interfaces on Human Factors and Interactive Computer Systems*. New York, USA: Ablex Publishing Corp., 1984, pages 1–28. ISBN: 0-89391-182-8. URL: http://dl.acm.org/citation.cfm?id=2092.2093.

[35] B. Shneiderman and C. Plaisant. *Designing the User Interface*. 5th edition. Pearson Education, 2010.

[36] M. F. Smith. *Software prototyping*. McGraw-Hill, 1991.

[37] *Steps in a Design Thinking Process*. 2017. URL: `https://dschool-old.stanford.edu/groups/k12/wiki/17cff/steps_in_a_design_thinking_process.html`.

[38] O. Taylor. "Pango, an open-source Unicode text layout engine". In: *25th Internationalization and Unicode Conference*. 2004.

[39] O. Taylor. *Pango: internationalized text handling*. 2001.

[40] K. Thoring and R. M. Müller. "Understanding the Creative Mechanisms of Design Thinking: An Evolutionary Approach". In: *Procedings of the Second Conference on Creativity and Innovation in Design*. DESIRE '11. Eindhoven, Netherlands: ACM, 2011, pages 137–147. ISBN: 978-1-4503-0754-3. DOI: `10.1145/2079216.2079236`. URL: `http://doi.acm.org/10.1145/2079216.2079236`.

[41] B. Victor. "Inventing on principle". Keynote given at CUSEC 2012. Jan. 2012. URL: `https://vimeo.com/36579366`.

[42] T. Warfel. *Prototyping*. Rosenfeld Media, 2011.

ISBN 978-3-86956-464-7
ISSN 1613-5652