Visual Enhancements for Code Examples to Support the Knowledge Exchange Between Programmers

by Maximilian Wendt
Abstract

Software development is an endless problem-solving activity and programmers regularly use online media channels to ask each other for advice. In particular, socially enabled media channels like the question & answer website Stack Overflow changed how programmers communicate and coordinate, and how they produce and consume content on the Web. Nowadays, a vast body of software development knowledge is documented in form of code examples. But acquiring the knowledge by reading code and logically reason about how they work makes non-trivial examples hard to understand. The aim of this work is to develop visual enhancements that support programmers in the knowledge exchange with code examples. Through a design science research approach using interaction design methods visualizations were developed and evaluated that allow programmers to see the execution and simultaneously inspect the state in order to gain a better understanding of how the underlying system of the code example works. Professional programmers participated in a workshop and found the visualizations to be helpful in the comprehension process.
Acknowledgments

I owe thanks to my supervisor Daniel Spikol for supporting me during this work with valuable advise, feedback, and motivation; Bahtijar Vogel for reading and commenting on this report; as well as my programming colleagues for testing and evaluating the prototypes.

Foremost and above all, I would like to thank my parents who enabled me to undertake my studies at Malmö University in the first place, and my girlfriend for her patience and support during that time.
Chapter 1

Introduction

In recent years, online media channels used by programmers are not only a way to communicate anymore but are becoming indispensable tools for them to effectively do their work (CHISEL (2013)). Software development\(^1\) is an endless problem-solving activity (Naselli et al. (2012)) and when an unexpected programming-related problem is encountered the first intuition is to search for a solution online. This is often times quicker due to the fact that it is very unlikely that a problem is so unique that no other programmer has had it before. Quite the opposite, chances are high that somebody else already had a similar problem, solved it and shared the solution online for others to use. For example, investigating an unexpected error in a program can take considerably longer because a programmer has to first find out what caused the error before being able to take measures to resolve it. Whereas a search on the Web takes only a few minutes and will most probably result in finding helpful information that enables a programmer to resolve it much quicker. It is therefore not surprising that a previous study which observed how programmers use resources during software development found out that programmers spent an average of 19\% of their programming time to mine the knowledge that exists on the Web. According to the study the time was used for three main reasons: just-in-time learning, clarification, and remembering difficult things (Brandt et al. (2009)).

One socially enabled media channel that is frequently accessed by a large crowd of programmers on a daily basis is the question and answer (Q&A) website Stack Overflow\(^2\). Since its launch in 2008, it has become a popular medium for mining, curating and distributing a growing software development knowledge. It allows anybody to ask and answer questions in many technical domains\(^3\) but especially programmers are using it to ask for advice on programming-related problems. Treude et al. (2011) found out that the most frequent types of questions on Stack Overflow are how-to and discrepancy questions, where the former is asking for instructions (just-in-time learning, remembering difficult

\(^1\) The term software development refers to the process of writing and maintaining code. At least in this report, it is not understood in any broader sense. Software development is therefore interchangeable with programming.

\(^2\) http://stackoverflow.com

\(^3\) http://stackexchange.com/sites#technology
things) and the latter is asking about some unexpected behavior the person asking the question wants explained (clarification). Thus, typical questions are, for example, ‘how to implement something’ or ‘what does a certain error mean’. When a question was answered successfully, that answer gets marked as ‘accepted’ by the person asking the question. This is only one of many advantage of using Stack Overflow over other online resources as it leads to a shorter search process because content is getting pre-ranked based on community expertise and it is not mixed with other irrelevant content (Treude and Filho, 2012).

Most of the time the accepted answers consist of two inseparable elements: code examples and the accompanied explanation (Nasehi et al., 2012). Both, code examples and explanations seem to be necessary characteristics of answers and can be regarded as the preferred format of programmers to share programming-related knowledge on Stack Overflow. Explanatory text provides the context or links to other sources while code examples are the essential information programmers are usually looking for. Often times, code examples are customized solutions which sometimes can, as a side effect, be copied and pasted into one’s source code and directly work or need only minor refinements (Treude and Filho, 2012). Despite being quicker, this is also convenient and programmers are relying more and more on the knowledge that is collectively curated by other programmers in the software development community.

1.1 Problematization

Code examples on Stack Overflow are written as text. This is only natural since almost all programming languages are text-based but also reasonable because text is quickly and easily created, changed, or removed. However, once code examples become part of the documented knowledge they get rarely changed, but potentially consumed thousands of times by programmers with varying programming skills. For that task a strictly textual representation makes understanding how code examples work harder than it has to be (Edwards, 2005). The only option a programmer has is to read it deliberately and simultaneously imagine how a computer would execute it. And when code examples are non-trivial, constraining them to a textual representation makes acquiring the necessary information a rather complex task. It is not possible to just execute the code, debug it using print statements or break points, as it would be possible with the code one is working on inside an IDE.

1.2 Research Aim and Goal

Pretty printing and indentation of block statements are currently the only graphical enhancements of code examples on Stack Overflow (see figure 1.1). Both are important and

---

4 Visual programming languages are not widely adopted, yet.
5 Despite reading it on a computer who in essence is only doing that: executing code
6 Integrated development environment
Figure 1.1: A typical answer on StackOverflow with pretty printed code snippets accompanied by explanatory text.
highlight syntactical aspects in code which makes code more readable. But, for reasons explained in detail later, a better understanding of the inner workings of the code example could be gained by visualizing not only the code but how the code executes. Therefore, the goal of this thesis is to find a way to build upon the existing process of producing and sharing these code examples, and design and test a novel program visualization approach that goes beyond the constraints of the normal textual representation that exists at the moment. The aim is to visually enhance the code examples in a way that lets programmers focus on how the code executes without logically reason step by step about everything written in code. It must be directly applicable by users on Stack Overflow and extend the existing knowledge transfer between programmers that is centered around sharing customized code examples in order to answer questions. The research question is therefore defined as follows:

RQ: How could code examples be visually enhanced to support the knowledge exchange between programmers on Stack Overflow?

1.3 Purpose

This thesis is a pilot study and the purpose is to revisit the rationale behind the current state of sharing software development knowledge in online media channels that exist today. Is the convenient and quick way of creating an ever growing amount of software development knowledge and its open and instantly accessibility the only benefit of using the Web as a medium? Or are there missed opportunities when looking more closely at the way knowledge is represented and used by programmers every time they develop software? In the beginning the Web was built for the single purpose of sharing text-based information between researchers. Then images were supported, followed by a period of third-party plugins that enabled formats like video, audio, and animations (i.e. Adobe Flash). Nowadays, modern Web browsers support all those formats natively including advanced graphical formats like scalable vector graphics (SVG). Thus, the question is if the medium is really used to its full potential?

1.4 Limitations

According to a recent survey of users on Stack Overflow there is a growing demand to be able to understand and write JavaScript code. For the last four years, most questions are concerning the use of JavaScript. Due to this fact and to make this research doable, this thesis is only focusing on the dynamic object-oriented programming language JavaScript. More specifically, only a limited amount of features were implemented to show the feasibility and test the usability of this novel program visualization approach.

7 It mimics the way code is visualized in most code editors or IDEs.
8 http://caniuse.com/#search=svg
By no means, does this research claim to produce a complete implementation. Yet, due to the conceptual design of the system it can be used as a basis for more sophisticated visual enhancements (see chapter 7.4 for inspiration and thoughts about future work).

1.5 Organization of This Document

The thesis is guided by a design science research methodology. The rest is therefore organized as follows. Chapter 2 presents the related research and theories necessary before being able to design an artifact. Chapter 3 introduces the design science process. Chapter 4 explains how the initial requirements were established. Chapter 5 describes the conceptual design of the system and important technical aspects. Chapter 6 covers two workshops used to evaluate early prototypes. Chapter 7 demonstrates and discusses the final artifact implemented according to the system design. Finally, chapter 8 concludes the thesis, discusses ethical considerations and future work.
Chapter 2

Background

Programmers are constantly working with code, they are fixing bugs, adding new features, enhancing working code, or changing legacy code. As mentioned before, software development is an endless problem-solving endeavor, and to get advice on how to accomplish any of these tasks programmers regularly mine a vast body of knowledge that exists on the Web. This chapter describes the role of social media on the software development process and how the lack of improving the visual perception of code in supporting programmers with program comprehension leads to a program visualization approach in order to deal with the inherent complexity of code examples.

2.1 Social Media in Software Development

An important role of media in software development has always been the transfer of knowledge between stakeholders (Storey et al., 2014) and the demand to communicate using the latest media has always been a part of software development. From 1968 to the present day almost any kind of media channel was used by programmers. For example, as early as the mid-90’s programmers started to use socially enabled media channels, similar to how blogs, wikis, podcasts, and discussion groups are used today, to transfer knowledge amongst each other (Storey et al., 2014).

According to Treude and Filho (2012) it is not only the vast body of knowledge, but the advances of social media that introduced effective mechanism for a large crowd of programmers to curate content on the Web. Some media channels were specifically designed and developed out of a perceived need by the software development community. As mentioned before, one of the currently most popular examples is Stack Overflow which was build out of the necessity to fulfill the need to more effectively ask and answer questions in a technical domain. Not only enabled it new forms of collaboration but enable programmers to exchange knowledge in a new, more efficient way. The exchange happens collectively in communities and almost instantly on demand with a median first response time of 11 minutes after the questions was posted (Mamykina et al., 2011). It is built around nine design features heavily influenced by social media mechanisms: voting, tags, editing, badges, karma, pre-search, Google is UI, performance, and critical mass
(Treude et al., 2011). All features taken together are used to collaboratively pre-filter and pre-rank content which considerably improves the likelihood to find relevant content. And before even evaluating any answer in order to determine whether or not it provides the necessary information the programmer can take into account meta-information that is especially valuable to filter out useless answers like: which user posted it; how good a reputation does the user have who gave the answer; or what is the rating of the answer from other users; etc.. Before Stack Overflow Q&A was normally done in technical forums with threaded discussions where useful information was usually mixed with irrelevant content.

Of course, Stack Overflow is popular in the software development community because of its ability to successfully fulfill a desired need (ask and answer questions more efficiently, facilitates to find qualitative code examples), but there also seems to be a paradigm shift taking place in software engineering right now resulting from the integration of such social features into the software development process.

2.1.1 The Social Programmer

Together with other researchers of the CHISEL group, Storey (2015) has been studying whether or not social tools are fundamentally changing the way programmers develop software. According to her, the reason for the shift are three fundamental changes in software engineering that are happening right now. The first trend is that, when comparing programs in the past with programs nowadays, the former used to be quite small, whereas the latter are very large and complex. This is because today they are made up of interconnected components and interconnected systems, which makes it pretty much impossible to understand how the ecosystem works as a whole. Coming out of this situation is a second trend, which is the development of socially enabled tools that allow programmers to connect and communicate with each other in order to share information, either in a project or in a community, at a scale that was unthinkable just eight years ago. This led to the third trend which is the recognition of the modern developer, who is, against some common stereotypical characteristics, very social and highly connected with other people.

The concept of the social programmer was proposed in 2012 in a position paper titled “Programming in a Socially Networked World: the Evolution of the Social Programmer” (Treude and Filho, 2012). Essentially, it is trying to grasp the way programmers use social tools in the software engineering process and what effects it might have, or already has, upon the practice and behavior of programmers when software development becomes a massively distributed activity. One of the key concerns is the questions of what will be the attributes of a ‘good’ programmer in the future – somebody who is an expert with a deep understanding of programming and software engineering principles, or somebody who can leverage and synthesize the programming community to achieve the same results?

\[1\] Before Stack Overflow or GitHub (another popular social coding platform).
2.2 Media as a Thinking Tool

Social media changes how programmers communicate and coordinate, and how they produce and consume content (Treude and Filho 2012). These new social features are advancing the organization of knowledge. But especially on Stack Overflow where code examples play a crucial role in the knowledge exchange and customized solutions are shared on demand almost instantly, no effort has been made so far to fully utilize the medium’s graphical capabilities to facilitate the knowledge transfer between the stakeholders. Will social programmers who focus on reusing content from those sites will have a good enough understanding of the inner working of the software they develop (Treude and Filho 2012)?

2.2.1 The Visual Perception of Code

Reading and comprehending code is a cognitive process that depends on the human visual system and Conversy (2014) rightly points out that ‘as with any visual scene, the performance of programmers reading textual [...] programs depends on their performance in perceiving the visual features presented on the screen". Some previous work was concerned with visual cues in the textual structure of the code, for example, studying the effect of indentation lengths (0-, 2-, 4- or 6-space indentation) (Miara et al., 1983) or identifier naming conventions (i.e. camelCase or under_score) (Sharif and Maletic, 2010) on a programmer’s performance in reading code. But such work does not go beyond the textual representation and merely formats the text itself. On the other hand, there is work that use graphics to augment the text. For example, one work was using graphic design principles for enhancing code using typography, typesetting, page composition mixed with symbols and diagrammatic elements, and metatext to make code more readable, understandable, appealing, memorable and maintainable (Baecker and Marcus, 1986). In both cases, the visual perception of code has been studied, but that research dealt mainly with ‘aesthetics’ or personal preferences’. Petre (1995) suggests that we first have to ‘understand the precise contribution graphical representations can make for the job at hand". He argues that there is a common misconception that graphics are often times superior to text. Since one purpose of code is to present information clearly and unambiguously, a textual representation is potentially better because it derives precision of expression from a small, fixed vocabulary that achieves range of expression by regular combination of vocabulary elements. But Petre (1995) also points out that there are contributions that only graphical representations can make and that is that they 'complement perceptually something also expressed symbolically" and he mentions the notion of secondary notation which uses ‘layout and perceptual cues (elements such as adjacency, clustering, white space, labelling, and so on) to clarify information (such as structure, function, or relationships) or to give hints to the reader" that might otherwise be less accessible. What is meant by that is that programmers can not only read code line by line and word by word, but are able to

---

2 The term cognition describes conscious mental activities such as thinking, understanding, learning, and remembering. (Source: [http://www.merriam-webster.com/dictionary/cognition](http://www.merriam-webster.com/dictionary/cognition))
recognize patterns in the text, allowing them to use the secondary notation to scan the code more effectively beforehand.

Ignoring other reasons for why code is represented as text, most of the research was conducted more than 20 years ago and there are no major changes in how code is represented nowadays, which brings up the question whether focusing on the visual perception of code can even bring significant improvements for the knowledge transfer through code examples.

2.2.2 A Medium for Understanding Systems

In a talk at the MIT Media Lab in 2013 titled ‘Media for Thinking the Unthinkable’ Victor (2013) asked the question of what a new medium for understanding systems would look like. He emphasized the importance of getting away from pencil-and-paper thinking. Especially in programming where programmers sit in front of a computer, work in a written language and logically reason step by step about everything written in code. The code is the structure of the system where as the behavior of the system is what the program is doing. To him the idea to be able to interact with the behavior of the system is of upmost important because “we are normally interacting with the structure of the system and that becomes the systems. When you are programming, you are working in code, you are thinking about code, you are interacting with the code, you are starring at the code all the time. The code becomes the program. But code does not matter. What matters is what the program is doing. So we need representations where we are focusing on the behavior of the system. We are interacting with the behavior of the system. We are starring at the behavior of the system. The behavior of the system becomes what it is all about.”

Thus, the trick to be able to visually and interactively explore the behavior of a system is to find a visualization that adequately represents the system and reproduce that.

2.2.3 Understanding Code Examples

It is complexity that makes understanding systems difficult. Brooks (1986) argued that complexity is an essential property of software entities and that complexity mainly comes from a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. He believed that the hard part of building software is not how the software is described (i.e. writing a program in a certain programming language), but to specify, design, and test the conceptual construct that underlies a program. He was talking about large-scale systems, but even though code examples are at a lower scale the constructs are the same. One important difference, though, is when weighing the concepts against each other it is mainly algorithms and the relationship between data items that lead to complexity in code examples. Huge data sets and mul-

---

^A program is defined as “a set of statements that can be submitted as a unit to a computer system and used to direct the behavior of that system” (Myers 1990).
tiple function invocations are not a big concern because they seldom play a role in the knowledge transfer with code examples.

In a follow up paper, Moseley and Marks (2006) explained that there are two main characteristics that are the cause of complexity in those interlocking concepts: state and behavior. They also discussed a widely-adopted general approach that can be used to eliminate them where they are accidental by nature. The approach is referred to as informal reasoning, also known as bottom-up (Storey et al. 2000), and the idea behind it is to attempt to “understand a system by examining it from the inside. The hope is that by using the extra information available, a more accurate understanding can be gained” (Moseley and Marks 2006). And since the constructs that causes the complexity in large-scale systems are conceptually the same as in code examples, informal reasoning as a general approach can be used to facilitate the comprehension process of code examples as well. This might seem trivial, but it can be used as the basic principle guiding the design of visual enhancements.
Chapter 3

Research Methodology

The visual enhancements of a code example using a program visualization approach can not be observed a priori. It is something that does not exist and is not yet in use [Krippendorff (2007)]. Quite the opposite, as it is a design issue and is concerned with how knowledge could be exchanged in the future. Therefore, in order to be able to say anything about how the knowledge exchange between programmers can be supported by such an artifact it needs to be developed. Subsequently, data can be generated and gathered which can then be used to answer the research question.

Thus, this thesis uses a design science approach which is a research approach for solving problems at the intersection of people, organizations, and technology ([Hevner et al.] [2004] [Peffers et al.] [2008]) or in other words “seeks to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts” ([Hevner et al.] [2004]). More specifically, it adapts the Design Science Research Methodology (DSRM) developed by [Peffers et al.] (2008) enriched by interaction design methods.

3.1 Research Process Model

Designing a visual enhancements for code examples call for an iterative development before they can be used to answer the research question ([Hevner] [2007] [Krippendorff] [2007]). Evaluation is necessary during the construction phase up until the designed artifact satisfies the requirements of the problem it was meant to solve ([Hevner] [2007]). Therefore the research process model proposed by [Peffers et al.] (2008) was chosen and adapted to the scope of this thesis. It now consists out of four steps (see figure 3.1) and whose application is summarized as follows:

Identify problem and explain motivation. The current state of mining, curating, and distributing a growing software development knowledge on Stack Overflow was analyzed in order to identify and frame the problem, as well as explaining the motivation for the research. See section 1.1, section 1.2 and section 1.3.

Define requirements of the problem. This involved conceptually breaking down the problem into requirements before designing and developing an artifact that provides
a potential solution. Requirements are traditionally classified into two kinds of requirements: functional requirements and non-functional requirements (Preece et al., 2011). The functional requirements say what the visual enhancements of code examples should do for the programmer and the non-functional requirements say what constraints there are on the design of the system and its development. In this case, the requirements are derived partly from the analysis of a successful solution to a similar problem (see section 4.1) as well as partly from the results of an online survey (see section 4.2) which was filled out by 12 professional programmers (see section 7.2 for ethical consideration about the validity of the results of the survey).

**Design and develop an artifact to demonstrate and evaluate a solution.** The goal of designing and developing an artifact is to demonstrate and evaluate its feasibility and usability. In this case the artifact is an instantiation (Hevner et al., 2004) that tries to support the knowledge exchange between programmers (see chapter 6). In two workshops, qualitative evaluation methods are used to improve upon the artifact and to see whether or not the artifact fulfills the functional requirements that are dependent on human factors.

**Communicate results.** The process ends with a discussion of the final artifact, especially on how it fits into the existing way of contributing to the collective software development knowledge, and why it is relevant. Finally, the last step is a reflection on the research process itself.

![Figure 3.1: The research process model adopted from Peffers et al. (2008).](image)
3.2 Data Gathering

According to a DSRM, the utility of the artifact needs to be rigorously demonstrated (Hevner et al., 2004). Qualitative evaluation methods were chosen since the utility of visually enhancing the representation of code examples is dependent on human factors such as visual perception and cognitive processes. Also, qualitative data is richer and more valuable insights can be gained in an early design phase (Cooper et al., 2007). The only exception was made for establishing the initially requirements. There an online survey was used to gathered quantitative data.

**Initial requirements: online survey** Using an online survey it was possible to reach out to 12 professional programmers leading to a more representative data for the initial requirements. Another reason for gathering data online was that the survey could be conducted simultaneous to the literature review and the analysis of the successful example – the other source for the initial requirements.

Once the first artifact was developed it was evaluated in two workshops using a controlled settings with a range of qualitative interaction design methods. The goal was to gather data from programmers about the effectiveness and usability of the artifact in supporting the task of reading and understanding code examples. The following interaction design methods were used during the study:

- **First iteration: focus group discussions** A focus group consisting of four professional programmer from an online agency was used to discuss the first artifact (see section 6.1). The goal was to find out how programmers perceive the usefulness of a single page of code’ enhanced by live output of data, line highlighting and input elements to see and control the behavior.

- **Second iteration: semi-structured interviews, usability testing & questionnaire** Five professional programmers were individually interviewed, then asked to take part in a usability testing that was screen-recorded, before filling out a short questionnaire asking about user satisfaction using the second iteration of the artifact (see section 6.2). The goal was to find out how programmers perceive the usefulness of visually enhancing code examples. During the usability testing scenarios were used to mimic the way programmers would search and use code example on Stack Overflow to solve programming-related problems. While using the code example they were encourage to think-aloud in order to find out what they thought while using it. The screen-recording were analyzed afterwards using a content analysis technique.

3.3 Research Context

The sample group for the online survey was chosen according to the ability to program. All participants were formally educated in computer science (or some related study pro-

---

1 All qualitative methods were taken from Preece et al. (2011)’s book “Interaction Design"
gramme such as computer science and media) and had been working for a few years within the broader software development industry. This is an important prerequisite in order to gather valid data for establishing the initial requirements. The participants were reached via Email or Facebook and asked to fill out the survey because they were living and working in different cities across Germany (including Freiburg, Frankfurt and Stuttgart).

The two workshops took place within one online agency in Germany (Freiburg) with up to five professional programmers that were currently working there full-time. Again, all programmers had a formal education in computer science or computer science-related programme and at least a few years of working experience (see table 3.1). All were also familiar with Stack Overflow and were using it every day at work.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Gender</th>
<th>Programming experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>M</td>
<td>13 years</td>
</tr>
<tr>
<td>P2</td>
<td>M</td>
<td>18 years</td>
</tr>
<tr>
<td>P3</td>
<td>M</td>
<td>5 years</td>
</tr>
<tr>
<td>P4</td>
<td>M</td>
<td>5 years</td>
</tr>
<tr>
<td>P5</td>
<td>M</td>
<td>9 years</td>
</tr>
</tbody>
</table>

Table 3.1: *Overview of participants that attended the workshops.*
Chapter 4

Establishing Requirements

As mentioned in the last chapter, a design science research approach uses an artifact to answer a research question. This means, as a first step requirements need to be established in order to guide the development of such an artifact. The basis for the initial requirements are the results from an online survey with professional programmers and the analysis of a successful solution to a similar problem. This chapter summarizes that process and defines the initial requirements of visual enhancements of code examples.

4.1 Analysis of a Successful Solution

Before defining the initial requirements an analysis of how another popular online media channel solved a similar type of problem through visual enhancements is used as inspiration. The media channel is GitHub\[\footnote{http://github.com}\] and the problem is how to support programmers in exchanging knowledge about changes in code.

Looking at figure 4.1 the problem is effectively solved by enhancing the normal representation of code and adapting it to the task of comparing two versions of the same file. Using visual enhancements programmers can see directly which lines changed and how the code looked before and after the change. By hiding all unaffected lines it is also possible to see multiple changes simultaneously even in files with many lines of code – note that this is optional and hidden lines can be expanded or collapse if necessary. Programmers are able to instantly reason about what effects those changes would have on the behavior of the program. In comparison, without any visual enhancements it would be much harder to find changes in those files. Programmers would need to simultaneously scan, read, and compare two versions next two each other before even being able to tell what really changed. This considerably increases the cognitive load on the person and makes it an unnecessary complex task.
Figure 4.1: Highlighting code changes in two versions of the same file on GitHub.
4.2 Results From an Online Survey

An online survey early in the research process gave some insights into what aspects of a ‘single page of code’ are useful for the comprehension of a program. The survey was online accessible for the whole period of writing this thesis and consisted of three parts and a total of ten multiple choice questions. The first part was concerned with the act of reading and comprehending code, the second part looked for preferred ways of documenting code, and the third part asked about the attitude towards openly sharing knowledge with other programmers. The link to the survey was sent to programmers of an online agency in Germany and other former programming colleagues of the author. In the end, 12 out of 17 programmers replied and filled out the survey. The following paragraphs summarize the responses.

1. Understanding code

Complexity, abstraction, and representation makes code hard to read The answers showed that complexity (i.e. resulting from the amount of lines of code) seems to be one of the main reasons that makes a ‘single page of code’ hard to read and understand - 92% of the respondents selected that property. Followed by issues with abstraction (i.e. using abbreviation for variable names) and representation (i.e. symbolic notation and layout) selected by half of the respondents.

Programming is a way of thinking 75% of the respondents think that programming is a way of thinking rather than a rote skill, meaning that it is a skill that can not be learned through repetition alone, but the performance of a programmer is rather dependent on the representation of the code or the program.

2. Documenting code

Code needs documentation The responses show that the documentation of code is seen as an important aspect of writing code according to 84% of the respondents - 42% of the respondents even strongly agreed. Especially other programmers’ comments inside code are regarded as helpful when trying to understand what the program is doing - 59% of the respondents agreed strongly.

Comments should describe functionality, dependencies and intentions According to 92% of the respondents inline comments should describe the functionality of the program (what the program does). Another 67% say that comments should describe the intentions the programmer had in mind while writing it (what the program is meant to do) and 58% want comments to describe dependencies it has on other parts of the program (what the program needs in order to work).

---

2 A ‘single page of code’ in the context of writing code in a text editor or IDE
3 All questions as well as responses can be found in the appendix (see chapter A)
Comments can not effectively describe limitations, behavior and dependencies

The answers also showed that 55% of the respondents think that a combination of code and comments is not the best way to communicate limitations (what the program can not do), as well as it is insufficient in communicating behavior (how the program works) or dependencies (what the program needs in order to work) - both selected by 45%.

3. Sharing code

Information about legacy code is searched for online, in comments, or face-to-face

The answers showed that when in need of more information about legacy code, 75% of the respondents search in online communities such as Stack Overflow. 67% search in documentations that are distributed online such as, for example, the official website of a framework such as jQuery[^1]. Also, 67% of programmers search for comments in the source code itself or they ask other programmers directly in person. This confirms the trend that most of the programmers are relying on software development knowledge on the Web (see section 2.1).

Knowledge about code is shared openly Without exception, all programmers would share their knowledge about code with any co-worker or other programmer.

4.3 Initial Requirements for Visual Enhancements

For the problem of supporting programmers with exchanging knowledge using code examples the same basic idea from section 4.1 is used: facilitate the specific task at hand by representing necessary information more appropriate. This time the task is to support comprehension. The idea is applied to the results from the online survey (see section 4.2) to inform the initial requirements as follows:

Reduce cognitive load The online survey showed that complexity seems to be the main reason that makes code hard to understand and that programming requires a certain way of thinking. As described in section 2.2.3 the root of complexity in code example comes from the interlocking concepts of state and behavior. This complexity should be reduced. The visual enhancements should support programmers to see and interact with state and behavior in order to use informal reasoning (see section 2.2). In other words, they should reduce the cognitive load by making procedural and declarative information more humanly accessible.

Extend the possibility to document code The online survey also showed that programmers seem to find documentation of code through inline comments useful. These comments are used to explain certain aspects of code in a natural language. It is sufficient for documenting the functionality, dependencies and intentions of the

[^1]: http://jquery.com/
code, but natural language is poor in describing limitations, behavior and dependencies. Thus, the visual enhancements should enable programmers to document aspects of the code difficult with natural language alone.

**Extend and support existing code examples** Code examples with visual enhancements should be producible in the same way as code examples are written today. The effort to write them with the added feature of the enhancements should be possible with little overhead. Because the goal is to support the knowledge exchange on Stack Overflow it must be based upon the same technologies and must match the way code examples are produced there. Additionally, it would be desirable to be able to enhance older code example that are already part of documented software development knowledge that exists on Stack Overflow.

The initial requirements for the visual enhancements of code examples are separated into functional requirements and non-functional requirements. The former requirements are concerned with the human factors and interaction design issues where as the latter are about the system design and technical implementation. The following list summarizes the final initial requirements:

1. Support reading and comprehension of code examples (*functional*)
   - (a) **FR1**: by enabling the programmer to readily scan the execution.
   - (b) **FR2**: by enabling the programmer to readily inspect the state.
   - (c) **FR3**: by enabling the programmer to readily test the behavior.

2. Be applicable in online media channels (*non-functional*)
   - (a) **NFR1**: by using native Web Technologies.
   - (b) **NFR2**: by extending the existing ways of producing code example on Stack Overflow.

All requirements together built the conceptual foundation for the system design. The functional requirements **FR1 - FR3** need to be evaluated repeatably with real programmers (see chapter [1]) and the non-functional requirements **NFR1 & NFR2** are being taken into consideration during the technical implementation in the following chapter.
Chapter 5

System Design

This chapter describes the design and development of visual enhancement for code examples and is addressing the non-functional requirements NFR1 & NFR2 that were established in the last chapter. The focus is on the technical implementation that takes a string of text (the code examples) and generates the visual enhancements. It is important to mention, though, that everything that is explained in this chapter is the most recent design of the system. See chapter 6 for details about the evaluation process of earlier designs and chapter 7 for discussion about how it answers the research question.

5.1 Exemplary Code Examples

There are three exemplary code examples[1]. Two of them are taken from a popular book titled “Eloquent JavaScript”[2] that teaches how to write programs in JavaScript (Haverbeke, 2015). The reason for that is that it takes away the ambiguity of having to deal with personal styles of writing code and lets the research focus on the visual enhancements rather than textual peculiarities. They are written in a ‘best practice’ of declaring variables, functions, etc. The same style of writing was used for a simpler third exemplary code example. Everything is written in pure JavaScript[3].

The difference between the three examples lies in the inherent complexity of the code example. As seen in section 2.2.3 complexity arises when there are many states and varying behavior.

Fibonacci The first exemplary code snippet is rather trivial and should be the easiest to read and comprehend. It is a function that takes a number as a parameter and returns the first $n$ numbers of the fibonacci sequence depending on the parameter. Understanding the state should be simple enough to be manageable in the head and the behavior should be straightforward even when the execution of statements is read only textually.

---

[1] The code examples can be found on the project website: http://mxwendt.github.io/clusters/

22
INI Parser The second exemplary code example should be something in between. It is a single function that parses a longer string and creates a nested object structure out of different section in that string. The computation deals with more complex states and the behavior is not immediately obvious which make it harder to comprehend than the Fibonacci example.

Electronic Life The third exemplary code example is a non-trivial program and should have the highest complexity of all three. It is a complete program and even though small in that sense it still has the most lines of code compared to the other two exemplary code example. That is also the reason why it should be harder to read than the other two examples because the program needs to be broken down into smaller parts in order to be able to reason about what it does. The fact that multiple smaller parts all have varying states that influence each other increases the complexity and should therefore be the hardest to comprehend.

5.2 Technical Implementation

The technical implementation deals with the process of turning a string of text (the code) into the native representation on the Web: the standard document object model (DOM). The process involves preparing the text, collecting data while parsing and interpreting it before finally rendering it as a visually enhanced code example in a browser. Every part is relying only on standardized Web Technologies.

5.2.1 Extending Code Example Annotations

A textarea element is the standard way of allowing users to input longer parts of text on a website. And since code is text written directly by the programmer, this is what is used by online media channels (addressing NFR2).

As described before in section comments are naturally used by programmers to share information that is not expressible with code alone. For such cases programmers annotate it using a natural language. This way, documenting and sharing information about functionality, intentions, or dependencies (meaning its effects on other parts of the code) is done in an effective way. However, commenting in the same way about its state and behavior, with varying data, and its resulting functional limitations, using text in a natural language alone seems complicated. Visual enhancements can help a reader, but how are the necessary additional information that is needed for the enhancements of the code example conveniently added in the first place? The idea is to extend the rules of annotating function parameters as defined by the JSDoc markup language.

http://eloquentjavascript.net/09-regexp.html
http://eloquentjavascript.net/07_elife.html
For further detail and the source code of the artifact the reader is referred to the public project repository on GitHub under https://github.com/mxwendt/clusters/
http://usejsdoc.org

23
an API documentation generator for JavaScript inspired by the industry standard of
documenting source code in Java.

To produce code example for the knowledge exchange, the programmer writes the
code and the comments at the same time. That way missing information that is needed
in order to generate the visual enhancements can be added in a natural way, simply
following the rules of annotation. For example, enhancing the code of the fibonacci
function requires a value for the size parameter so that the function can be executed
and interpreted. Thus, the value is defined in the comments which get parsed and make
the value accessible.

Extending Function Parameters

Only values that are not already defined in the code example need to be added. The
following section describes how the notation of function parameters are extended for each
standard type in JavaScript.

The level of granularity is that of standard types in JavaScript: numbers, strings,
booleans, arrays, or objects. These are written in the exact same way as they are written
in the code. For example, to add the information that ‘a function takes a parameter with
the value of eight’ the value would simply be 8 or if the parameter is an array the value
would be written like [0, 1, 2, 3, 4, 5] and so on.

Boolean values: Parameter values of the type Boolean are extended by adding a value
of either true or false which is required and is used as the initial value.

```javascript
//
/**
 * @param {Boolean} foo = true
 * @param {Boolean} foo = false
 */
function fubar(foo) {}
```

Number values: Parameter values of the type Number are extended by three addi-
tional values, which are all number values and are all required. The first value is
used as the initial value, the second value is used as the minimum value, and the
third value is used as the maximum value.

```javascript
//
/**
 * @param {Number} foo = 8, 0, 10
 */
function fubar(foo) {}
```

String values: Parameter values of the type String are extended by adding additional
values that are strings. The first value is required and is used as the initial value.

---

[^: http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html]
All following values are optional. Thus, there is always one initial value with an variable amount of alternative values.

```javascript
/**
 * @param {String} foo = "blah-blah", "bluh-bluh", "blih-blih"
 */
function fubar(foo) {}
```

**Array values:** Similar to string value, parameter values of the type *Array* are extended by adding additional values that are arrays. The first value is required and is used as the initial value. All following values are optional. There is always one initial value with an variable amount of alternative values.

```javascript
/**
 * @param {Array} foo = [0, 1, 2, 3, 4, 5], [0, 2, 4, 6, 8, 10], [3, 2, 1]
 */
function fubar(foo) {}
```

**Object values:** Parameter values of the type *Object* are extended by particularly specifying all values that are actually used inside the function. This involves a higher level of granularity, specifying each value as one of the previous four types: boolean, number, string, or array. All rules as explained for those types apply here as well. What comes additionally is that the parameter name reflects the object property using the standard dot notation in JavaScript.

```javascript
/**
 * @param {Object} foo = {}
 * @param {Object} foo = null
 * @param {Object} foo = undefined
 * i.e. dot notation, one level, all types:
 * @param {Boolean} foo.thud = true
 * @param {Number} foo.bar = 0, 0, 10
 * @param {String} foo.baz = "blah-blah"
 * @param {Array} foo.qux = [0, 1, 2, 3, 4, 5]
 * @param {Object} foo.ysr = {}
 * i.e. dot notation, multiple levels:
 * @param {Boolean} foo.thud.bar.baz = true
 */
function fubar(foo) {}
```
5.3 Program Visualization Approach

There are four steps in the program visualization approach that turns a string of text into a visually enhanced code example: parse, beautify, visualize execution, and visualize state. In the end the textual code is enhanced by additional visualizations as in figure 5.1. The visual enhancements should enable a programmer to see and understand the whole execution – which lines get executed and in which order – and what the state of the program is at each step. The slider above the time-series plot can be used to quickly inspect a state at a given point of the execution. Scrubbing the slider one can sense and reason about the behavior without the cognitive overhead of remembering each state at each part of the execution. The slider next to the parameter value in the state can be used to change the initial value of the parameter variable.

![Visual enhancements for the Fibonacci code example.](image)

**Parse Code Example** A parser is used to analyze the code (that comes as a string of text) in a syntactical way in order to facilitate the writing of a small JavaScript interpreter. An open source JavaScript parser called Acorn\(^\text{10}\) goes through the code and creates and abstract syntax tree (AST) object as specified by the EStree spec\(^\text{11}\). The community standard specifies the core node types include statements nodes, like for example function or variable declarations – each line is some type of statement. The resulting AST is used by the JavaScript interpreter to executes simple code examples while at the same generating data about the execution and state that can later be used for the visualizations.

**Beautify Code Example** The code in its initial form is not formatted in a consistent

---

\(^9\) This is an example for a Number parameter (see section 5.2.1)
\(^10\) https://github.com/ternjs/acorn
\(^11\) https://github.com/estree/estree

26
way and is therefore beautified using an open source JavaScript beautifier\footnote{http://jsbeautifier.org/}.

![Figure 5.2: Beautified code for the Fibonacci code example on the right.](image)

**Visualize Execution** The D3 JavaScript library\footnote{https://d3js.org/} is a visualization library for the Web following a data-driven approach to DOM manipulation \cite{Bostock2011}. It is used for the visualization of the execution, placing a dot for each statement in the text on the same line but spread over time on the x-axis. In graphic design this representation is called a time-series plot and is one of the most frequently used forms of visualizing data over time. Since understanding execution of code is essentially understanding a sequence of statements – meaning what statements get executed at what time – a time-series plot seemed to be an effective way to show the execution of the code over time. The execution is quickly scanned and it is easier to see when something gets executed or whether some lines get not executed at all. It is possible to recognize and find parts of the code that get called more than other parts by simply scanning clusters of dots. Additionally, coloring the dots according to the type of statements (i.e. if statements or while statements with red and green dots for truthy or falsy statements) it is possible to recognize control-flow structures such as loops or if-else cases or see where new variables get defined (blue dots).
Visualize State  To visualize the state the UI library RactiveJS\footnote{http://www.ractivejs.org/} is used which, similar to D3, binds data to templates. Each statement is a step and each statement that alters the state of the program is saved as a snapshot of that state at that step of the execution and the templates get updated accordingly.

Since the beautifier, the parser, the visualization library, and the UI library are all written in JavaScript themselves, the visual enhancements of code examples can be done directly
in the browser and on the fly. The string of code to be visualized can be provided through an input DOM element or it can be taken from existing DOM elements already representing the code example in a textual format (addressing \textbf{NFR2}).
Chapter 6

Evaluation

The novel program visualization approach for code examples was developed through a series of prototypes over the course of six months. In the beginning, two concepts for enhancing a ‘single page of code’ were discussed, which inspired the next round of prototypes that was used to enhance three specific code examples similar to way they could be found on Stack Overflow. The code examples differed in complexity and were tested with five professional programmers against their effectiveness and their usability in facilitating comprehension – thus the artifact’s utility (Hevner 2007). The general goal of all visual enhancements was always to test how good they support informal reasoning by visualizing state and behavior (see section 2.2.3). This chapter describes the results from those two workshops.

6.1 Workshop I: A Single Page of Code

The representation of a ‘single page of code’ was visually enhanced in order to find ways to deal with the complexity during the act of reading and comprehending the code. One goal was to simplify the process of informal reasoning by externalizing parts of the mental simulation that otherwise take place in the head of a programmer (a feature labeled state during the discussion, addressing FR2 & FR3). The other goal was to reduce the number of possible scenarios that need to be considered simultaneously (a feature labeled focus, addressing FR1). Input elements, live output and line highlighting were chosen as means to achieve both goals. The hope was that it would allow for a tighter feedback loop by reducing hidden internal state and simplifying the case-by-case mental simulation of behavior – “if this variable is in this state, then this will happen — which is correct — otherwise that will happen — which is also correct” (Moseley and Marks 2006). Figure 6.1 shows the ‘single page of code’ with the features state and focus activated.

1Four of the programmers already tested the two initial ideas.
Figure 6.1: Seeing all visual enhancements simultaneously when the state of the system is changed: all affected lines are highlighted in yellow while unaffected line in between are hidden; showing live output in red for each statement that changes the state of the system.
6.1.1 Setup

The first workshop conducted a focus group discussion to gain insights through an early prototype. The aim was to evaluate the initial concept which tries to support a programmer when dealing with the complexity of code examples. The prototype consisted of a ‘single page of code’ that was visually enhanced to (a) enable a programmer to readily see the data and therefore reduce the cognitive load and (b) give visual cues in order to improve the perception of the code itself. Four professional programmers attended for a 90 minutes workshop.

During the group discussion the prototype was presented and the participants were asked several question to engage them in the discussion and to gain qualitative insights about the effectiveness of visual enhancements addressing the requirements. Besides general feedback coming up naturally, the most important questions were (a) what the first steps are and what is hard when they try to comprehend a page of code like this and (b) what they though about the features labeled state and focus.

6.1.2 Results

The key comments from the evaluation of how visual enhancements should be used are summarized in the following paragraphs. The complete notes can be viewed in the appendix B.0.1.

1. When asked about the first steps of trying to comprehend the code the participants mentioned the following:

   **Highlight the first statement** 3 of the 4 participants mentioned that they first try to find the first statement. It would be important to find the start of the execution.

   **Show the execution of the computer as a whole** 2 of 4 mentioned that in the beginning it is important to first get an overview of what is important. They scan the code for hints before starting to read it line by line.

   **Provide an overview that leverages the visual system** 2 of 4 mentioned that the complex structure of code is a problem. They mention difficulties arise from too many lines of code and the nesting of statements which forces a programmer to jump around in the code as well as making the comprehension of the underlying system hard.

2. Comments about the perceived usefulness of the two features labeled state and focus were the following:

   **Feature state shows potential** Visualizing the effects a line of code has on the state of the program directly next to the code was seen as an improvement to not seeing the state at all, but the positioning it right next to the code was not perceived as optimal because it creates confusion of what is code and what is state.
Feature focus was very good Highlighting every line effected by a change in the program through any of the input elements was seen as very good. Unfortunately the feedback on this feature was very sparse.

3. Other recommendations by group were the following:

Build upon existing IDE features Participant were missing some code visualization they were used to from working with existing IDEs. These features should be the foundation that these new features built upon.

Test visual enhancement with more complex example The new examples should be more complex functions with multiple parameters and effects, i.e. use a standard function like substring().

The representation showed promise, and the presented features were seen as useful and effective with supporting the task of reading and understanding a ‘single page of code’. The insights influenced the next iteration in the design science process and were not used to adjust the requirements, but used to design the next round of prototypes.

6.2 Workshop II: Code Examples

The developed system (see chapter 5) was used to visually enhance three code examples (see section 5.1) in order to test the refined visualizations. Again, the goal was to see if the visualizations support the participants in gaining an understanding of how the code examples work. This time the execution of statements was visualized as a time-series plot using dots (see section 5.3, addressing FR1), and the state was placed separate to the code (addressing FR2). Each part – code, execution, state – had its own area and nothing was integrated into the code examples itself as it was the case with the previous prototype. By using a slider displayed above the time-series plot it was possible to step through or jump around each statement. Optionally, it was possible to click on a dot to jump to a specific statement. The Electronic Life example also offered a feature to change input values for some parts of the code. Being able to explore the execution and state simultaneously, it was hoped to enable the participant to understand the behavior (addressing FR3).

6.2.1 Setup

The second workshop consisted of three parts: interview, usability testing, and questionnaire. It was held with five professional programmers. Each programmer was invited individually to take part in a session that started with a semi-structured interview, used a usability tests to gather data about the usefulness of the visual enhancements of code examples, and ended with a short questionnaire that asked about user satisfaction. The interviews were used to prepare the programmers for usability tests and asked them about their opinion and adoption of using Stack Overflow as a source of knowledge. During the usability test the participants were encouraged to talk aloud about what they
were doing and what they were thinking at the moment. The usability test were screen-
recorded (video and audio) in order to be able to analyze afterwards. A categorization
scheme was derived from the set of functional requirements about the artifact’s capability
to support the comprehension process of code examples. It was used for the content
analysis of the transcripts of the screen-recordings. Finally, the programmers filled out
a short questionnaire consisting of five questions about the visualizations.

6.2.2 Results From the Think-Aloud Protocols

The result from the content analysis of the think-aloud protocols were coded with the
following categorization scheme:

1. Verbalizations show evidence of supporting the programmer to understand the code
example because:

   (a) it was possible to more readily scan the execution.
   (b) it was possible to more readily inspect the state.
   (c) it was possible to more readily test the behavior.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Positive verbalizations</th>
<th>Negative verbalizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan the execution</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>Inspect the state</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Test the behavior</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.1: The relation between positive and negative verbalizations of how the visualiza-
tions support programmers with the comprehension of the code examples.

Scanning the Execution

Being able to visually scan the execution was perceived as useful. Being able to see what
lines of code get executed, what comes first and what comes last, and especially being
able directly spot lines of code that not get executed at all. It reduces time until one is
able to reason about the inner workings of the code example, for example it is possible to
see control-flow like loops or if-else structures more clearly. Also, because it augmented
the preferred comprehension strategy of going through the code line by line. The wish
was stated to test it on an even more complex example than the INI Parser example
which had the most complex visualization of the execution.

Inspecting the State

The main benefit of visualizing the state was that it helped to reduce false assumptions
about the values of particular variables within the code. It supported the programmers
in not overlooking important information while going through the code. It helped com-
prehension by not necessarily having to understand every statements, but being able to
skip over complex computation like regex expression within if-statements. Especially, being able to see how a value of a return variable was changing before being actually returned by the function was mentioned as very helpful.

Testing the Behavior

Testing the behavior was seen as helpful when a programmer was able to test out extremes like using minimum or maximum input values and just ‘play around’ and see how it behaves. It was especially useful for complex logical expressions statements in the code and being able to change only parts of the expression and inspect the changes it had on the computation. Setting arbitrary limitations in order to test behavior could be seen as problematic in certain cases, for example, one participant was irritate by a minimum input value that was only defined to make the visualizations possible, but had no other meaning for the code.

General Feedback

The visualizations needed a little adaptation time and could lead to confusion and unnecessary added complexity, but once it was explained a little bit it was generally perceived as a positive enhancements to code examples. For example, one participant wanted to be able to optionally use it and actively activate it if needed. That it was only supporting each function separately was not stated as a drawback, quite the contrary, one of the participants even broad up the idea before it was explained. And, one participant emphasized that it would be a useful feature on Stack Overflow. When asked about the additional work of writing the annotations for the code examples all participants agreed that it would be a minor overhead in comparison for the benefits it brings. However, this is only true for simple primitive types like booleans, numbers, short strings, and extending the standard way of using the JSDoc annotation was not received as difficult. Other features were also discussed from time to time that not directly related to the research, but show a general interest and usefulness of supporting a programmer in reasoning about code.

Results from User Satisfaction Questionnaire

At the conclusion of the workshop the participants were asked about their opinion of the visual enhancements. All questions are on a 5-point scale, with 1 labeled ‘Strong disagree’, 3 labeled ‘Neither’, and 5 labeled ‘Strong agree’. The complete questions were as followed:

1. The visualizations are clear and easy to understand.
2. The visualizations allow me to scan the execution.
3. The visualizations allow me to inspect any specific point in the execution.
4. The visualizations allow me to test the behavior by offloading the computation needed to understand the state to the computer.

5. I find this helpful.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Age</th>
<th>Gender</th>
<th>Clear and Easy</th>
<th>Scan Execution</th>
<th>Selective Inspection</th>
<th>Offload Computing</th>
<th>Helpful</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>-</td>
<td>M</td>
<td>*** • •</td>
<td>• • •</td>
<td>• •</td>
<td>• • •</td>
<td>• •</td>
</tr>
<tr>
<td>P2</td>
<td>-</td>
<td>M</td>
<td>• • •</td>
<td>• • •</td>
<td>• •</td>
<td>• • •</td>
<td>• •</td>
</tr>
<tr>
<td>P3</td>
<td>-</td>
<td>M</td>
<td>• • •</td>
<td>• • •</td>
<td>• •</td>
<td>• • •</td>
<td>• •</td>
</tr>
<tr>
<td>P4</td>
<td>-</td>
<td>M</td>
<td>• • •</td>
<td>• • •</td>
<td>• •</td>
<td>• • •</td>
<td>• •</td>
</tr>
<tr>
<td>P5</td>
<td>-</td>
<td>M</td>
<td>• • •</td>
<td>• • •</td>
<td>• •</td>
<td>• • •</td>
<td>• •</td>
</tr>
</tbody>
</table>

Table 6.2: Workshop II post-survey questionnaire. All questions are on a 5-point scale, with 1 labeled ‘Strong disagree’, 3 labeled ‘Neither’ and 5 labeled ‘Strong agree’.
Chapter 7

Discussion & Conclusion

This thesis was set out to answer the research question of how code examples could be visually enhanced to support the knowledge exchange between programmers on Stack Overflow. To answer that question, a novel program visualization approach was developed that can produce visually enhanced versions of code examples as they are frequently shared on Stack Overflow. After two iterations it was able to produce visual enhancement that satisfied the initial requirements\textsuperscript{1}. These requirements were established through an online survey and the analysis of a successful solution to a similar problem, both based on a literature review of relevant topics concerning the visual perception of code, essential complexity in code examples and program comprehension strategies.

7.1 Research Results

The main findings come from the second workshop which was conducted in a controlled environment due to the early phase of the artifact \cite{Preece et al., 2011}. During the usability tests with five professional programmers it became clear that a short adaptation phase is necessary. All programmers, intuitively started to read the code when asked to try to understand the code example. Often times ignoring the visual enhancements and only using them afterwards as a confirmation to their previously gained understanding. While only reading the code all participants naturally used a form of informal reasoning. They started by reading the comments, then function name before mentally going through the code step by step reasoning about the execution. When starting to use the visual enhancements, almost all of them used it to go through the code more or less step by step and simultaneously inspected which lines where executed and how the state changed as an effect of that. Sometimes jumping between two steps several times to make sure they understood what has changed in the state.

After using the visual enhanced code examples for 20 minutes on average all five professional programmers gave positive feedback and confirmed that the visual enhancements support them to visually scan the whole execution and inspect the state for any

\textsuperscript{1}Please refer to chapter 5 for examples of how the visual enhancements look
given step. The visualization was clear and easy to understand resulting from the fact that it aligns with prior strategies of reading and comprehending code. Visualizing the underlying system and enabling them to interact with the system was perceived as a useful and effective way to gain a better understanding of the behavior.

Since this visualization of the underlying system is only one of many possible ways to visually enhance code examples other representations need to be tested going forward to confirm these preliminary findings which are based upon limited requirements. Nonetheless, it should provide a basis and could inspire further research. Especially due to the design of the developed program which is able to parse and execute JavaScript code to create arbitrary data that then can be visualized.

7.2 Research Validity

Some issues concerning the validity of this research must be noted. It is important to point out that the data collected from the online survey was filled out by 12 people and is therefore not seen as representative, but rather as indicative. However, the author still considered it valuable data since it was only used to define the initial requirements which were being tested in an iterative fashion and could be adopted after each evaluation phase if necessary.

Also, it needs to be mentioned that data gathered from real programmers interacting with early prototypes was limited to five programmers from one online agency the author is currently employed at. It is therefore hard to generalize and results from the evaluation with real programmers should be seen as preliminary. Nonetheless, it is hoped to provide valuable insights for future work in that area. Especially since a group of five programmers is not an unusual size for a software development team.

7.3 Code Sharing Sites

The Web as a medium to share programming-related knowledge, especially knowledge in form of code, is used by other socially enabled media channels as well. For JavaScript in particular code sharing sites exist, that usually consist of an editor to write HTML, CSS and JavaScript, and a live preview area that updates instantaneously as the code gets edited.

---

2For example, step through the whole execution or jumping to specific steps (i.e. often times the last step), or changing input values.
  https://codepen.io/hello/
4https://jsbin.com/help/getting-started
5https://jsfiddle.net/
Figure 7.1: The Fibonacci code example on the code sharing site Codepen with the editor on the left and the live preview area and console output on the right.

Being able to see the code and the output at the same time and see how the output changes as the code gets edited is a valuable feature. It invites users to play around and figure out how the code works by making changes in the code and observe how the output is affected. But what is missing are the steps *in between*. What cannot be seen is how the computer incrementally builds up the output. This is the essential difference of the visual enhancements developed in this thesis over such code sharing sites and their real-time output.

To get similar features as provided by the visual enhancements, especially those that are similar to a debugger\[7\] programmers would need to fall back onto full programming environments that run inside a Web browser. Often times referred to as Cloud IDEs, they provide the same functionality as Desktop IDEs, with additional social features such as sharing and collaborating on code in real-time. Those IDEs are not especially useful to quickly ask other programmers outside the development team for advice since they are not dedicated media channels to openly share knowledge. Communication is integrated into a development environment for programmers to improve collaboration. Apart from that, these IDEs provide more sophisticated debugging features in regard to visualizing

\[7\]https://de.wikipedia.org/wiki/Debugger
state, but they are lacking the ability to provide an overview of the execution which the visual enhancements can provide through the time-series plot.

7.4 Future Work

Of course, as with any software, there is always potential to add new features that could explore further visualizations of code, data, or algorithms. For this artifact in particular, the data visualization of the state is still very basic and would benefit from further design and evaluation iterations. It is currently comparable to the level of a simple visual debugger.

Future research would involve field testing directly on Stack Overflow to analyze the usefulness within a real world setting. This would be a logical next step, but would bring about new challenges like the question on how to integrate such an artifact on a proprietor media channel like Stack Overflow.

One possible solution could be to develop it into a browser plugin for that purpose.
Chapter 8

Conclusion

The objective of this thesis was to develop and evaluate a technology-based solution to the problem of how the exchange of software development knowledge could be supported within socially enabled media channels as they exist today. The focus was on Stack Overflow, where the preferred way to exchange knowledge is achieved by sharing code examples amongst the stakeholders. It discussed issues with the current representation of code examples and its influence on exchanging knowledge amongst programmers before an artifact was developed that demonstrated the feasibility, benefits, and usability of using visual enhancements to support the knowledge transfer. The proposed solution consisted of a novel program visualization approach written in JavaScript that not only reproduced the visualization of the code but also enhances it by visualizing the procedural (execution) and declarative (state) information of the code which allowed programmers to visually and interactively explore the system.

It could be shown that the benefit of using the Web as a medium, in particular socially enabled media channels like Stack Overflow, is on the one hand the open and instant access to a vast software development knowledge collectively organized by the expertise of a global community. But that it is equally useful to utilize the unique ability of the medium to represent information in new formats that successfully leverage the human visual system to improve understanding, or in other words: use vision to think.


Margaret-Anne D. Storey. How Social Media Shape a Participatory Culture in Software Engineering, 2015. URL [https://www.youtube.com/watch?v=PTlV-aE8RvQ](https://www.youtube.com/watch?v=PTlV-aE8RvQ).


Appendix A

Online Survey

The results from the online survey with 12 professional programmers from Germany which formed the basis for the initial requirements.

A.1 Understanding Code

**Question 1:** How would you describe your programming skills?

<table>
<thead>
<tr>
<th>Answer Choices</th>
<th>Responses</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0/12</td>
<td>0.00%</td>
</tr>
<tr>
<td>Little</td>
<td>0/12</td>
<td>0.00%</td>
</tr>
<tr>
<td>Weak</td>
<td>1/12</td>
<td>8.33%</td>
</tr>
<tr>
<td>Some</td>
<td>8/12</td>
<td>67.67%</td>
</tr>
<tr>
<td>Strong</td>
<td>3/12</td>
<td>25.00%</td>
</tr>
</tbody>
</table>

**Q2:** What makes ‘a single page’ of code hard to read and/or understand? *Note: A ‘single page of code’ in the context of writing code in a text editor or IDE*

<table>
<thead>
<tr>
<th>Answer Choices</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction - What do these words mean?</td>
<td>6/12</td>
</tr>
<tr>
<td>Calculation - What is the program doing? What happens when?</td>
<td>4/12</td>
</tr>
<tr>
<td>Representation - <em>i.e. the symbolic notation and layout: syntax highlighting;</em> code indentation; ...</td>
<td>6/12</td>
</tr>
<tr>
<td>Complexity - <em>i.e. the amount of lines of code</em></td>
<td>11/12</td>
</tr>
<tr>
<td>Other (please specify)*</td>
<td>2/12</td>
</tr>
</tbody>
</table>

- Use of ‘short codes’ / abbreviations like ‘?’ for ‘if then else’.
- Names of variables.

**Q3:** I spent too much time during programming trying to understand legacy code (i.e. old code or written by another programmer).
Q4: Do you consider programming a rote skill (i.e. learned through repetition) or a way of thinking?

<table>
<thead>
<tr>
<th>Answer Choices</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rote skill</td>
<td>3/12 25.00%</td>
</tr>
<tr>
<td>Way of thinking</td>
<td>9/12 75.00%</td>
</tr>
<tr>
<td>Neither</td>
<td>0/12 0.00%</td>
</tr>
</tbody>
</table>

A.2 Documenting Code

Q5: Code documentation is important.

<table>
<thead>
<tr>
<th>Answer Choices</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong disagree</td>
<td>1/12 8.33%</td>
</tr>
<tr>
<td>Disagree</td>
<td>0/12 0.00%</td>
</tr>
<tr>
<td>Neither</td>
<td>1/12 8.33%</td>
</tr>
<tr>
<td>Agree</td>
<td>5/12 41.67%</td>
</tr>
<tr>
<td>Strong agree</td>
<td>5/12 41.67%</td>
</tr>
</tbody>
</table>

Q6: Other programmer’s comments inside source code help to understand what the code doing.

<table>
<thead>
<tr>
<th>Answer Choices</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong disagree</td>
<td>0/12 0.00%</td>
</tr>
<tr>
<td>Disagree</td>
<td>0/12 0.00%</td>
</tr>
<tr>
<td>Neither</td>
<td>1/12 8.33%</td>
</tr>
<tr>
<td>Agree</td>
<td>4/12 33.33%</td>
</tr>
<tr>
<td>Strong agree</td>
<td>7/12 58.33%</td>
</tr>
</tbody>
</table>

Q7: What should comments describe in order to be useful to programmers?
Q8: What do you think is hard to communicate using just code and comments?

A.3 Sharing Code

Q9: Where do you seek useful information about legacy code you are working with?

Q10: I would share my knowledge about code with any co-worker or other programmers.
<table>
<thead>
<tr>
<th>Answer Choices</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong disagree</td>
<td>0/12 0.00%</td>
</tr>
<tr>
<td>Disagree</td>
<td>0/12 0.00%</td>
</tr>
<tr>
<td>Neither</td>
<td>0/12 0.00%</td>
</tr>
<tr>
<td>Agree</td>
<td>2/12 16.67%</td>
</tr>
<tr>
<td>Strong agree</td>
<td>10/12 83.33%</td>
</tr>
</tbody>
</table>
Appendix B

First Workshop

The notes from the focus group discussion with four professional programmers.

B.0.1 Focus Group Discussion

Table B.1 lists opinions from the four participants about what the first step is when they try to understand code.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Comments</th>
</tr>
</thead>
</table>
| P1          | • Find the first statement, start of execution  
             • Understand interrelations/system |
| P2          | • Get an overview  
             • Understand vocabulary (variable names, ...) |
| P3          | • Find the first statement, start of execution, then follow the execution |
| P4          | • Scan the whole code and only focus on the important parts (skip unimportant parts)  
             • Start where the computer would start and go through like the computer would (interpret like a computer)  
             • Search for comments and read them when they help understanding the code at hand (not long comments usually used to explain a framework or similar; official documentation; context)  
             (Referring directly to the function Grid.isIsolated)  
             • Names and parameters are descriptive and tell you something about their intention  
             • Test values that work then test values that don’t work  
             • Mental unit testing |

Table B.1: Comments about comprehension strategies.

Table B.2 lists opinions from the four participants about what is hard to understand when trying to understand the code.
<table>
<thead>
<tr>
<th>Participant</th>
<th>Comments</th>
</tr>
</thead>
</table>
| P1          | • Nesting  
              • Complexity through many LOC |
| P2          | • Variable names |
| P3          | - |
| P4          | • The fact that it can not have different meanings like in a fictional story for example  
              • Jumping around in the code  
              • Which variables have which values? |

Table B.2: *Comments about what makes comprehension hard.*

Table B.3 lists opinions from the four participants about an feature which was labeled *state* during the discussion.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>• Improve the visualization of limitations/boundaries of data</td>
</tr>
</tbody>
</table>
| P2          | • It helps understand the functionality  
              • Inline descriptions are missing, i.e. What is a vector? What is a grid?  
              • Eventually extend the JSDoc notation by @return, if it does not lead to inconsistency  
              • Scalability could be an issue |
| P3          | • Quiet cool  
              • Tempted first to try to understand the extension before understanding the code  
              • Concerned whether the tool can be trusted |
| P4          | • Cool, very helpful  
              • Probably (really) helpful with more complex examples  
              • Emphasized the feature to only highlight values that are needed  
              • It would be really helpful when parts or all of it could be automated in the creation process of the visual enhancements |

Table B.3: *Comments about the experimental state feature.*

Table B.4 lists opinions from the four participants about an feature which was labeled *focus* during the discussion.
Table B.4: *Comments about the experimental focus feature.*

Table B.5 lists general feedback from the four participants that was interesting for the further development of the visual enhancements of code examples.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>• Very good</td>
</tr>
<tr>
<td>P2</td>
<td>• Very good</td>
</tr>
<tr>
<td>P3</td>
<td>• Very good</td>
</tr>
<tr>
<td>P4</td>
<td>• Very good</td>
</tr>
</tbody>
</table>

Table B.5: *General feedback.*

<table>
<thead>
<tr>
<th>Participant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>-</td>
</tr>
<tr>
<td>P2</td>
<td>• Vocabulary must be clear</td>
</tr>
<tr>
<td></td>
<td>• Code is not clearly structure, could be improved</td>
</tr>
<tr>
<td></td>
<td>• &quot;IDE is a kind of visualization of the code&quot;</td>
</tr>
<tr>
<td>P3</td>
<td>-</td>
</tr>
<tr>
<td>P4</td>
<td>-</td>
</tr>
</tbody>
</table>
Appendix C

Second Workshop

The goal of the second workshop was to find out if code examples can be visually enhanced to support the knowledge exchange between programmers on the web. The representation was enhanced to support the reading and understanding of the code examples, in particular the testing and reasoning about the behavior.

C.0.1 Interviews

The semi-structured interview was asking about the habits of mining the collective software development knowledge on the Web and searching for programming-related solutions.

- Do you search the Web for answers to programming-related problems?
- If yes, can you describe how you do it?
- Would you say that searching the Web has an effect on productivity? What effect?
- What kind of sites do you find especially useful?
- Do you use Stack Overflow? How often?
- What is your opinion on code snippets when reading a solution on Stack Overflow? Are they important or helpful?
- Do you use GitHub to learn from other programmers, i.e. to get an idea of how something works or could be implemented?
- Have you copied code snippets from Stack Overflow or code from GitHub into your source code when they solve your problem? How do feel about that?
- When have you decided that the code is useful to you and worth copying into your own code?
- Have you heard of program visualization? Do you know what it is?
C.0.2 Usability Tests

Scenario 1: Fibonacci

Imagine you are still a computer science student. You are sitting at home in front of your laptop and are having a hard time to do an assignment you have for your "algorithm and data structure 101" class. The assignment is to write a function that returns the first n numbers of the fibonacci sequence in JavaScript. You start searching the Internet to get you an idea how you could write this. You end up on GitHub where a user has a repository with classic algorithms written in JavaScript. Each algorithm has its own file. The one you are interested in is named fibonacci.js.

The following enhanced code example was provided:

```javascript
/*
 * fibonacci.js
 * (c) 2012 number
 * 
 * 
 * return (ret sequence was made but it was not good);
 * }
 * 
 * if (size < 2) {
 * return 'the request was made but it was not good';
 * }
 * 
 * while (count++ < size) {
 * next = first + second;
 * first = second;
 * second = next;
 * }
 * 
 * return result;
 */

function fibonacci(size) {
    var first = 0;
    var second = 1;
    var count = 2;
    var result = [first, second];
    if (size < 2) {
        return 'the request was made but it was not good';
    }
    while (count++ < size) {
        next = first + second;
        first = second;
        second = next;
    } return result;
}
```

Figure C.1: Enhanced code example of showing how to calculate the Fibonacci sequence in JavaScript.
Scenario 2: INI Parser

Imagine you are working on a computer program that needs to be able to be configurable across different platforms. You have read that you can use an INI file to accomplish this. Now, you don’t know how to parse the file and search on the Web for answers. You end up at Stack Overflow where another programmer had already had the same problem. You read the question and it seems to be what you’re looking for. The accepted answer has a code snippet that shows how you can parse an INI file using JavaScript.

The following enhanced code example was provided:

Figure C.2: Enhanced code example of showing how to parse an INI file in JavaScript.

Questions for Each Scenario

- Did the visualization help you to understand the code snippet? How?
  - Why or why not?
  - Can you explain how?
    - What information is useful?
    - What information is still missing?
    - What information is unnecessary to represent?
    - What information is better represented? What is worse?
    - Is the added information helping or disturbing?

- Is it helpful to see invisible aspects of a program in the comprehension process?

- Is it helpful to be able to scan the execution? Why?

- Is it helpful to be able to see the state? Why?

- Is it helpful to be able to see the whole behavior? Why?

The behavior refers to the way a program modifies its state over the time of its execution including the way it behaves to different data values.
• What do you think about the additional markup in the annotation?

• Does it feel natural to provide data in the comments? Do see any difficulties with that? Would you prefer a different way?

• Is it something you would do? Does it seem to be too complicated or not worth the extra effort?

• Would you see a benefit in visualizations on a site like Stack Overflow?

• Would you see a benefit in visualizations on a site like Stack Overflow?

• Would you see a benefit in visualizations on a site like GitHub?