# Communication Of Source Code Designs Through Sketching

Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

*by*
*Lukas Spychalski*

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Björn Hartmann

Registration date: Oct 31st 2012
Submission date: May 21st 2013

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

*Aachen, May 2013*
*Lukas Spychalski*

# Contents

# List of Figures

# Abstract

Understanding the design of source code and the mental model behind it is still a major problem for programmers. Many software visualization tools are designed to support programmers, but these tools are dependent on the underlying source code, and programmers need to know more than the source code can convey in order to understand it. Sketching is an established tool for ideation, exploration and communication and software developers use sketches frequently in different phases of the software development process to depict and convey different views and concepts of the system under development.

To aid the communication of source code designs, I introduce the functionality to integrate hand-drawn sketches into a software development environment and connect them to source code.

After an initial study of software architects and developers regarding the use of sketches in their everyday work, the fundamentals of a connection between source code and sketches are presented. Based on these fundamentals, a software prototype is developed that connects sketches and source code. For the purpose of evaluating the software prototype and its functionality, 32 participants were observed and interviewed in a user study. The results and implications of the user study as well as suggestions for future work are presented.

# Überblick

Das Verstehen von Quelltext und dem dazugehörigen mentalen Modell ist auch heute noch ein großes Problem für Programmierer und Softwareentwickler. Zwar steht ihnen eine große Auswahl an Zusatz- und Hilfsprogrammen zur Verfügung, die den Quelltext visualisieren und dabei das Nachvollziehen von Quelltext und seinem Aufbau vereinfachen sollen, jedoch können diese Programme nur das visualisieren, was bereits im Quelltext vorhanden ist, wobei Programmierer mehr benötigen um Quelltext zu verstehen. Das Erstellen von Zeichnungen ist eine beliebte Art der Ideenbildung, Weiterentwicklung von Ideen und Kommunikation. Programmierer und Softwareentwickler erstellen häufig Zeichnungen während des Entwicklungsprozesses um verschiedene Ansichten und Konzepte von Softwaresystemen zu visualisieren und zu erläutern.

Um den Aufbau des Quelltextes besser vermitteln und kommunizieren zu können, bietet der von mir implementierte Softwareprototype die Möglichkeit Zeichnungen in eine Softwareentwicklungsumgebung einzubinden und diese mit dem Quelltext zu verbinden.

Nach der Präsentation der Ergebnisse aus einer Erststudie, bei der ich Softwarearchitekten und Softwareentwickler beobachtet und befragt habe in Bezug auf die Verwendung von Zeichnungen im Arbeitsalltag, werden die Grundlagen zur Erstellung einer Verbindung zwischen Zeichnungen und Quelltext vermittelt. Darauf aufbauend wird die Entwicklungs des Softwareprototypen vorgestellt, der die Verbindung von Zeichnungen und Quelltext unterstützt. Um den Prototypen und seine Funktionalität zu evaluieren, wurde eine Benutzerstudie mit 32 Teilnehmern durchgeführt. Abschließend werden die Ergebnisse der Benutzerstudie sowie Anregungen zur Weiterentwicklung des Prototypen und der Funktionalität präsentiert.

# Acknowledgements

With these words I would like to sincerely thank everyone who supported me, first and foremost my parents and my wife.

Hiermit möchte ich mich bei allen bedanken, die mich unterstützt haben, vor allem bei meinen Eltern und bei meiner Ehefrau.

# Conventions

Throughout this thesis I use the following conventions.

*Text conventions*

Definitions of technical terms or short excursus are set off in coloured boxes.

**EXCURSUS:**
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

The whole thesis is written in American English.

# Chapter 1

# Introduction

This thesis addresses the communication of source code designs through sketching. The approach presented in this thesis is to connect sketches directly to source code in order to support the communication of source code designs. This chapter provides an introduction and is structured as follows:

**Section 1.1—"Thesis Context and Motivation"** gives an insight into the software comprehension process of programmers and the role of sketches in this context. Based on that, the motivation for connecting sketches with source code in order to communicate source code designs and support programmers during their software comprehension process is formulated and constitutes the foundation for my approach.

**Section 1.2—"Thesis Structure"** presents an outline of the thesis structure by briefly summarizing the individual chapters.

## 1.1 Thesis Context and Motivation

Understanding source code takes a lot of time and mental effort.

Understanding the source code of computer programs is one of the core software engineering activities (Singer et al. [1997], Ko et al. [2006], LaToza et al. [2006]) and is required in many situations, e.g., when a programmer maintains, reuses, migrates, refactors, or enhances software systems. Software developers must gather a variety of information in order to acquire knowledge about source code when trying to edit and maintain that source code: What is connected to what? Which changes affect the code elsewhere? How are design decisions scattered across the code? What is the rationale or history behind decisions? Who is the owner responsible for the code? These are some questions to which answers can be helpful while trying to comprehend a software program. However, the key to understanding and successfully editing source code is not only gathering knowledge about it, but rather putting this knowledge into use.

Understanding source code is building and refining mental models about its behavior. Research suggests three basic software comprehension strategies.

Müller et al. [1993] define software comprehension as "the task of building mental models of the underlying software at various abstraction levels, ranging from models of the code itself, to models of the underlying application domain, for maintenance, evolution, and reengineering purposes". Software comprehension models try to explain and describe the way programmers attempt to understand source code and researchers suggest three types of comprehension models:

- **Bottom-up comprehension models** propose that the understanding is formed by reading the source code and then chunking these low-level information and mentally grouping them into high-level abstractions. This is mainly the case when programmers have little or no knowledge about the underlying application domain. (Shneiderman and Mayer [1979], Pennington [1987], Détienne [2002])

- **Top-down comprehension models** suggest that programmers have knowledge about the application domain and that this knowledge is utilized by programmers to build expectations that are mapped onto the source code (Brooks [1983], Shaft [1992], Good et al. [1999]).

- **Combinations of the aforementioned models** are suggested as strategies pursued by programmers since findings show that programmers switch between comprehension models and strategies in response to external cues and stimuli (von Mayrhauser and Vans [1995], Letovsky and Soloway [1986]). In particular, Letovsky [1986] states that "the human understander is best viewed as an opportunistic processors capable of exploiting both bottom-up and top-down cues as they become available".

Although software comprehension models differ in their focus, they all incorporate four common elements that play an important role during the process of understanding source code designs:

All software comprehension models consist of four general components: External representations of the source code, knowledge in the head, a mental model and the assimilation process that constantly refines the mental model.

1. **External representations** are external views that support the programmer during the process of comprehension. Examples for external representations are software documentation, the source code itself, tools that offer additional information about the source code, and expert advice from other programmers with knowledge about the source code in question.

2. The **knowledge base** can be seen as the acquired knowledge gained before trying to understand the source code. This knowledge may be knowledge about the domain of application, programming standards and practices as well as experience. Shneiderman and Mayer [1979] divide the knowledge base into syntactical knowledge (language dependent, regards statements and basic units in a program) and semantical knowledge(language independent knowledge, enables formation of mental model). The knowledge base grows as the understanding deepens.

3. The **mental model** is the programmer's current understanding of the system, i.e., the internal, mental representation of a real system's behavior, organization, and internal structure.

4. The **assimilation process** is the strategy that the programmer employs in order to comprehend the source code. During this process the mental model is continuously updated using external representations, the

knowledge base and the current mental model that is about to be refined (Davies [1993]). One method for the assimilation process is forming hypothesis about the system's behavior which are refined and verified during the process (Brooks [1983]).

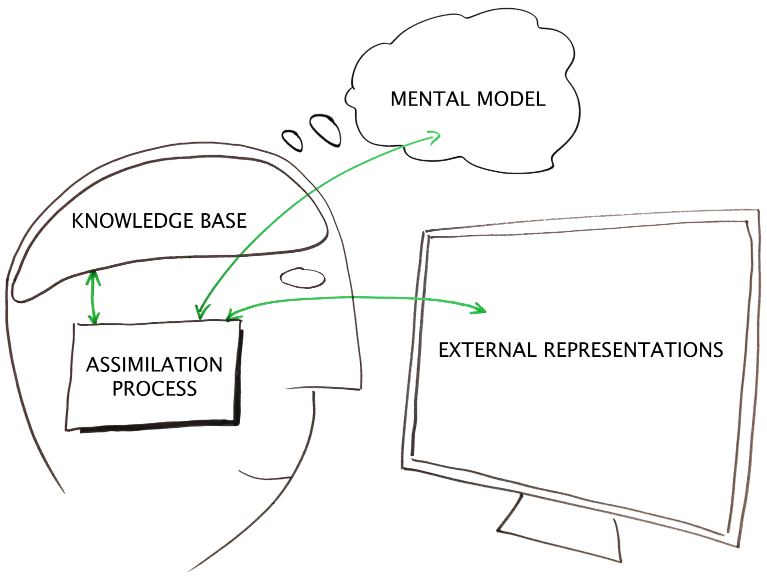


**Figure 1.1:** Components of software comprehension models: The mental model is constantly updated during the assimilation process using the knowledge base, external representations and the current mental model.

Since source code provides low-level information, additional external representations should provide higher-level abstractions.

External representations provide a facility to expand the knowledge of software developers and programmers and, thereby, refine the mental model of the source code. Since programmers need low-level details as well as high-level concepts according to the comprehension models, whereas the source code itself is a very low-leveled and detailed source of information, it seems standing to reason to provide high-level information about the structure and design, e.g., visualizations and drawings of the software architecture or data models.

Many tools provide external representations in the form of software visualizations.

Many different approaches and tools have been presented and developed in order to support the software comprehension process by providing external representations in the form of automatically generated software visualizations

(Eick et al. [1992], Bragdon et al. [2010], DeLine and Rowan [2010], Kurtz [2011a]). In essence, the visualizations represent high-level abstractions created on basis of the underlying source code. Clearly, the advantage of these tools is that the visualizations are created automatically and, therefore, without additional work effort of the user.

**SOFTWARE VISUALIZATIONS:**
*Software visualizations* are visualizations created with the support of *computer-based tools*. One kind of software visualizations are *source code visualizations* which are automatically created visualizations from underlying source code. Source code visualizations are created by *re-engineering tools* that operate on the source code. An example is shown in Figure 1.2. Another kind of software visualizations are *tool-based visualizations* that are not based on source code. These tool-based visualizations are created with *computer-based visualization tools* like Microsoft Visio.

Re-engineering tools operate on existing source code.

However, since the tools operate on the existing source code, they can only visualize what is already present in the source code. Hence, information about the source code design, that can not be conveyed via the source code itself are not provided by these tools.

**SKETCH AND SKETCHING:**
In this thesis a *sketch* is meant to be a hand-made drawing that represents the result of sketching. *Sketching* is to be understood as a visual thinking tool utilized in processes like ideation, exploration of alternatives and conversations with self or others. Sketching can be performed in an analog or digital way: The analog way includes tools like pen+paper or marker+whiteboard. The digital way includes tools like digital pens and graphic tablets operated with a stylus. An example is shown in Figure 1.3.

Hand-drawn sketches are a helpful tool to create, share and document knowledge.

Empirical research shows that software developers and programmers create hand-drawn sketches in order to understand existing code and form a mental model about its behavior (LaToza et al. [2006], Cherubini et al. [2007b], Walny et al. [2011]). Sketches have the advantage of conveying visuospatial ideas directly, using elements and spa-

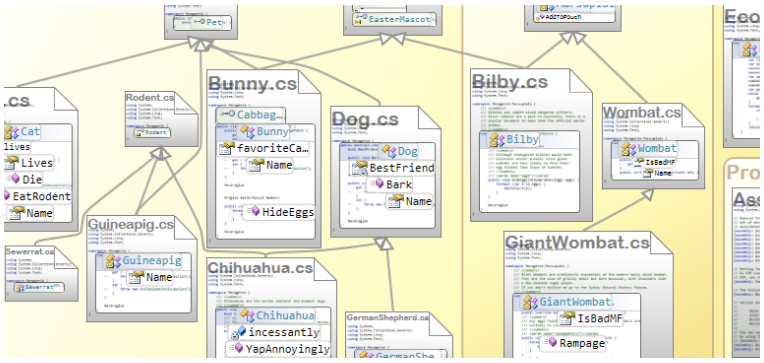


**Figure 1.2:** Example of a source-code visualization created with the re-engineering tool Code Canvas (DeLine and Rowan [2010]).

tial relations, e.g., on paper, to convey elements and spatial relations in the world: Expressing ideas in a visuospatial medium makes comprehension and inference easier than in a more abstract medium such as language (Tversky and Suwa [2009]). Sketching can be used to create, share and document knowledge about the software design and the underlying source code (Eppler and Pfister [2011b]). Visualizing knowledge with sketches is a powerful means of communication and can enhance conversations since it allows more immersive and creative collaboration (Eppler and Pfister [2011a]).

Sketching is a tool for ideation, exploration and communication.

Sketches are often the starting point of a software development project, since sketching is an inexpensive and forgiving way to explore a multitude of possibilities and quickly get an overview of the parameters and basic conditions of the project. A brief outline of the software architecture or the unpolished prototype of a user interface can be realized in a very short time and with very little effort, but with a lot of gain in terms of the direction the project should be heading. Moreover, it is used during the maintenance phase in which functionalities are refactored and additional, new features are designed.

Software developers need to know more than the source code can convey in order to understand it.

Entering the maintenance phase of an existing project, developers often need to grasp the mental model, design rationale or design decisions of the source code at hand. Developers involved in the development of that project may

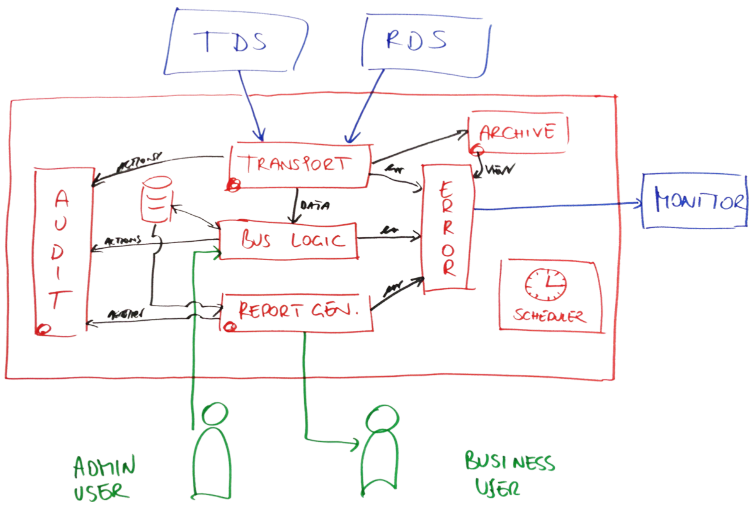


**Figure 1.3:** Example of a hand-drawn sketch.

just need a reminder or hint in order to regain the understanding. However, a new team member joining a software development project in its maintenance phase was not present while structural layouts were discussed and design decisions were made. But the sketches created during initial meetings and design sessions are external representations that can be helpful to the new team member and support the understanding of source code designs.

To put it briefly, hand-drawn sketches and diagrams are code independent and can be used to visualize thoughts and ideas that are not in the source code itself. However, the creation of hand-drawn sketches takes time and requires additional work effort, but sketches are created during the software development process anyway and are also archived (Walny et al. [2011]).

Sketches are code independent and can capture the mental model in a visuospatial way.

Based on the preceding line of argument, it seems standing to reason to introduce the ability to integrate sketches into the source code: The approach pursued in this thesis is to support the software comprehension process by introducing another external representation that incorporates hand-drawn sketches. In order to realize this aim, I introduce and evaluate the functionality to connect sketches to the corresponding source code in order to communicate source code designs.

Sketches connected to source code should provide an additional channel of information to communicate source code designs.

## 1.2 Thesis Structure

**Chapter 2—"Related work"** provides an overview of literature published in the context of sketches in general, sketches in the domain of software development as well as tools for software developers that deal with sketches and software visualizations.

**Chapter 3—"Initial Study"** presents my first-hand experience gained while observing two software architectural meetings and interviewing software architects and developers.

**Chapter 4—"Fundamentals"** provides basic ideas and definitions concerning the connection between sketches and source code.

**Chapter 5—"Prototyping"** introduces my software prototype that provides the functionality to connect sketches and source code. Moreover, I describe my design decisions in detail and explain certain aspects of the implementation.

**Chapter 6—"Evaluation"** describes the experimental setup and how I conducted the user study. Also, the quantitative and qualitative results gathered during the user study are presented.

**Chapter 7—"Summary and Future Work"** summarizes the results of this thesis and gives suggestions that should be addressed in future.

# Chapter 2

# Related work

This chapter offers an overview of literature published in the context of this thesis. The overview provides background information to understand the study, establishes the importance of this topic and justifies the approach taken in this thesis. This chapter is structured as follows:

**Section 2.1—"Sketches and Sketching in General":**
presents findings about sketching from research conducted in the field of cognitive psychology with the focus on domains like architecture, industrial and graphical design, as well as engineering.

**Section 2.2—"Sketches and Software Developers":**
narrows the focus to the software development process and presents findings on why and how software developers sketch in their everyday work.

**Section 2.3—"Sketches and Tools for Software Developers":**
introduces several sketching tools for software developers that aim to enhance the sketching experience.

**Section 2.4—"Visualization Tools for Software Developers":**
introduces several re-engineering tools that aim to support software developers by visualizing existing code.

**Section 2.5—"Consequences for this thesis":**
summarizes the findings and explains the influence on the development of this thesis.

## 2.1 Sketches and Sketching in General

Sketches are used to externalize thoughts and ideas to relieve the working memory.

Complex and detailed trains of thoughts can be a burden for the limited-capacity working memory. Sketches are a common way to relieve the working memory of that burden and, thereby, externalize thoughts and ideas. Hence, sketches are a visual representation of imagination: Sketches ensure that fleeting thoughts can be stored permanently. Moreover, sketches can convey spatial as well as abstract concepts that portray literal mappings, like buildings, or metaphorical mappings, like organization charts (Tversky [1999], Tversky [2002]). But sketches are more than just an external storage for ideas.

The ambiguity of sketches supports ideation and innovation by allowing different interpretations of a sketch.

Unlike a model that demands completeness, sketches are vague, partial, and eliminate detail that is irrelevant and distracting, while still capturing the essentials. That is why sketches are ambiguous by definition. This ambiguity contributes to the fact that different people can have different interpretations of the same sketch, but also that one person can have different interpretations in the course of time. But instead of being the source of confusion, this ambiguity seems to further innovation through reinterpretation, which can lead to the extension of thoughts and the discovery of new ideas (Tversky and Suwa [2009]). This process is described as backtalk of self-created sketches (Goldschmidt [2003]) or as a conversation designers have with their own sketches (Schön [1983]) and is an essential part of the design and ideation process.

Sketches are a powerful tool of communication and help with understanding.

Moreover, sketches can be used as a tool to guide and focus a conversation. Pointing at relevant parts on a sketch adds context and focus at the same time. Visualizing thoughts, knowledge and ideas with another person or even with a whole group will enhance the conversation and add a natural flow and pace to it. Using sketches and sketching techniques in a conversation allows more creative and immersive collaboration, which may lead to better listening, recollection and understanding of the issues discussed (Eppler and Pfister [2011a]).

Most findings focus on cognitive psychology.

All these findings about sketches are based on research conducted in the field of cognitive psychology with the focus

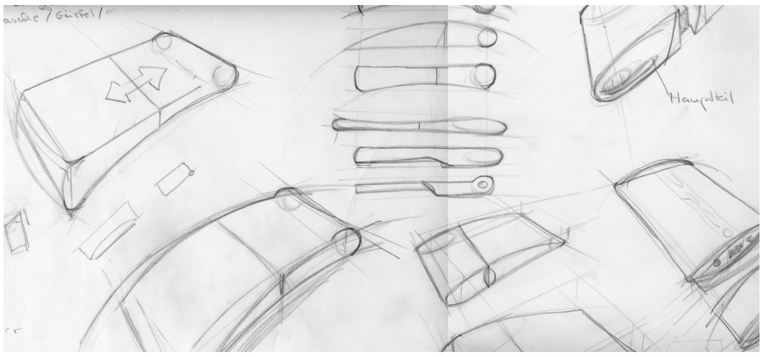


**Figure 2.1:** Sketches of a paper-prototype for a wearable computing device designed for blind people with an optical implant. The sketchy character enables creating multiple versions quickly and can lead to new ideas.

on domains like architecture, industrial and graphical design, as well as engineering. An interdisciplinary literature review on the benefits of sketching regarding the management of knowledge is given by Eppler and Pfister [2011b]. In their review the highlighted disciplines are psychology, design and computer science. Three categories emerged in which sketches are beneficial and support the findings presented so far. The three categories are knowledge creation, knowledge sharing, and knowledge documentation. However, the benefits are not put in contrast to other knowledge management techniques, but computer-based knowledge management solutions seem to be distractive and add an unnecessary information overload. In addition, many benefits of sketching were found in more than just one discipline. Which leads to the assumption that some of those benefits may be universal and, therefore, also may be applied to the software design and software development process.

Interdisciplinary findings suggest that the benefits of sketches are also applicable in the field of computer science.

## 2.2 Sketches and Software Developers

Spitballing, designing, creating, editing, refining, communicating and understanding are typical activities software developers perform frequently (LaToza et al. [2006], Walny

Software development is all about the source code.

et al. [2011]). Most of these activities are focused on the source code. But despite the fact that it is written in a human-readable programming language, source code seems not to be a suitable medium of communication between human beings, since understanding code written by someone else or self is a serious problem for software developers (Ko et al. [2006], LaToza et al. [2006]). Moreover, the lacking ability of source code to convey its own rationale or history is also identified as a serious problem: That is why software developers spend a lot of time understanding unfamiliar code and are trying to find information about it (Singer et al. [1997], Ko et al. [2006], LaToza et al. [2006]).

Conveying knowledge about the source code is a real problem.

Meaningful names for methods, variables, and files, as well as comments and documentation are helpful ways to share information and common knowledge that cannot be reflected well in the implementation itself. But adding comments or creating a good documentation requires additional expenditure of time besides the time that is already spent creating, editing or refining the source code. As a consequence developers rarely use or rely on documentation since they feel that it does not get updated frequently and often provides to much information that is poorly written. The "ugly truth" about documentation is that it is untrustworthy in large parts (Lethbridge et al. [2003]). Furthermore, several parts of knowledge about code are never written down and only exist in the head of the developers.

Knowledge about source code is mostly in the head of software developers.

The adopted way to get the needed information about unfamiliar source code is to ask other team members in short, but interruptive ad-hoc meetings, where the knowledge gets frequently visualized in transient form, i.e., mainly on whiteboards or paper (LaToza et al. [2006], Cherubini et al. [2007b]). The result is that these visualizations of mental models and the knowledge around source code have value after the day of creation (Branham et al. [2010]), but since the sketches are rarely written down and archived afterwards the knowledge is constantly rediscovered (LaToza et al. [2006]).

In order to get a better understanding of the working habits and practices of software developers in relation to sketches, the following subsections describe why and how software developers use sketches in their everyday work.

### 2.2.1 Why do software developers sketch?

Three scenarios were identified as very important with regards to the creation of sketches.

In order to identify the scenarios in which developers sketch Cherubini et al. [2007b] interviewed nine software developers and identified nine scenarios in which software developers sketch. Subsequently 400 Microsoft employees (81% software developers, 11% development leads, 5% architects, 3% others) were surveyed in order to get a deeper understanding of the nine identified scenarios. Three out of nine scenarios were mentioned as the most important with regards to creating sketches, whereas in the remaining six scenarios the importance of sketches was either very similar or exceeded by the importance of software visualizations:

1. **Understanding existing code** describes the process of examining the source code and its behavior as well as the usage of additional tools like documentation. The goal is to form a mental model and develop an understanding of the dependencies and relations.
2. **Designing/refactoring** describes the process of planning new or restructuring existing functionalities. Exploring new ideas and improving existing code also requires a deep understanding of the structure and the mental model behind the code.
3. **Ad-hoc meetings** are informal meetings of team members in groups of usually two and at most five. Missing, additional information about code or code-related topics are gained in these meetings that are almost never scheduled and, therefore, interruptions of other team member's work.

During the onboarding process new team members try to gain an understanding of unfamiliar source code.

Other identified scenarios are mainly derivations and combinations of these three main scenarios: Onboarding describes the phase in which a new member is joining a group of developers. Onboarding is a typical example for the process of understanding existing code. Missing information that cannot be gathered by self is then gathered in ad-hoc meetings in which senior team members explain existing code and create sketches to convey mental models of the code.

Software developers archive and reuse sketches.

Walny et al. [2011] also give a detailed view for the domain of software development and observed many similar areas of application in comparison to other disciplines that use sketches frequently. They followed sketches and diagrams in their lifecycle within a software development process to gain an insight whether and why sketches are used. The scenarios in which developers created and used sketches are very similar to those listed above. But, in addition they found that sketches get modified, copied and shared after their day of creation and also get archived for later use.

The findings presented in this subsection suggest that sketches have a value for software developers and are used in many different situations as a supporting tool in addition to code itself. The following subsection gives an insight on how software developers sketch.

### 2.2.2 How do software developers sketch?

Software developers create sketches mainly on whiteboards and on paper.

Software developers mainly sketch on paper or on whiteboards (Kurtz [2011b]) depending on the circumstances and anticipated size of a sketch. Whiteboards, however, were identified as the most adapted tool for producing sketches since the production costs of a sketch were reported to be very low. Developers can freely sketch, without being constrained by formal notations or specific standards. But with an increasing level of granularity the costs of creating a sketch increase and the creation of very detailed sketches is therefore less common (Cherubini et al. [2007b]). Making use of paper and whiteboards, as well as notebooks, printers, scanners, cameras, photocopiers, hand-held devices, tablet devices and PCs indicates that developers have established their own and individual workflows in dealing with sketches (Walny et al. [2011]).

Hand-drawn sketches dominate software visualizations.

In all scenarios described in subsection 2.2.1 hand-drawn, analog sketches predominated over both tool-based visualizations and source code visualizations created with re-engineering tools. Moreover, the scenarios understanding, designing/refactoring and ad-hoc meetings seem to use microscopic as well as macroscopic views on the code (Cherubini et al. [2007b]). Developers state that a macroscopic view, in which the high-level understanding is shown of how a feature should work, is useful for the documentation of software and that these high-level concepts have value even if they are not up-to-date, since they still provide enough important information (Lethbridge et al. [2003]). Developers further state that a macroscopic view cannot be depicted automatically by a re-engineering tool (Cherubini et al. [2007b]). Hence, especially in the three scenarios understanding, designing/refactoring and ad-hoc meetings more than 75% of the surveyed software developers agreed or strongly agreed that hand-drawn, analog sketches were important. Whereas re-engineering tools were all beneath the 25% mark (Cherubini et al. [2007b]).

In summary, findings showed that mainly informal notations were used and current re-engineering tools were unsuitable in many scenarios since they offered no help in externalizing the mental models of code. Sketches are mostly used for purposes like ideation and communication with others and self.

## 2.3 Sketches and Tools for Software Developers

Computer-based sketching tools omit the Design Flow.

Sketching with pen and paper or on a whiteboard is the perfect example for direct manipulation: "you draw what you want, where you want it, and how you want it to look" (Gross and Do [1996]). Computer-based sketching tools add the burden of dealing with the tool itself and thereby omit the freedom and fluidity that is provided without such a tool. Using computer-based tools instead of sketching on paper hinders creativity and shifts the user's focus to refining the sketch by concentrating on colors, fonts, and alignments (Wong [1992], Goel [1995]). But, at the same time computer-based tools offer certain advantages like editing, sharing, and digitally archiving sketches. In addition computers support 3D modeling, rendering and simulating as well as remote collaboration (Gross and Do [1996]). Therefore, approaches that combine these two worlds should be very appealing to software developers.

Tools should support the Design Flow.

An important guideline for tools that support sketching is, that they should respect the fluidity of sketching, i.e., the Design Flow (Dorta et al. [2008]) by letting the sketchers concentrate on the sketching task and not distract them with the tool (Csikszentmihalyi [1991]).

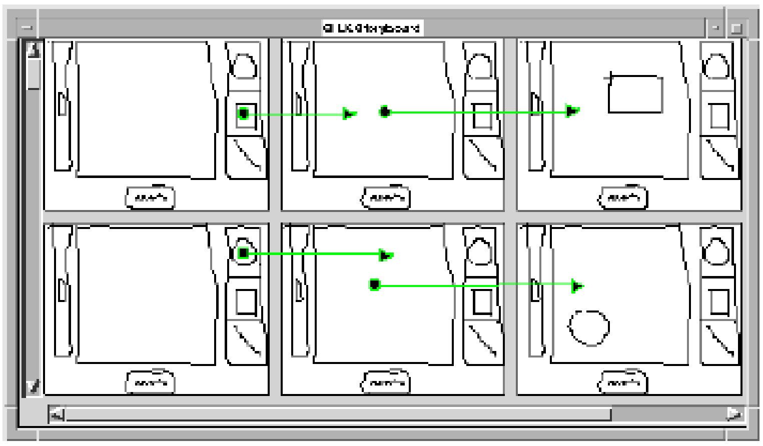


**Figure 2.2:** SILK introduced by Landay [1996] is an interactive sketching tool. Designers can quickly sketch an interface using an electronic pad and stylus, and SILK recognizes widgets and other interface elements. SILK supports the creation of interactive storyboards.

Digital sketching tools are not popular, due to poor sketch recognition and beautification.

In 1996 Landay introduced SILK, a tool for sketching and improving user interface prototypes, that could be operated with an electronic pad and a stylus. Landay envisioned that in the future user interface designers would create most of the user interface code with the help of tools like SILK (Landay [1996]), EtchaPad (Meyer [1996]), DENIM (Lin et al. [2000]), InkKit (Plimmer and Freeman [2007]), and PaleoSketch (Paulson and Hammond [2008]). These tools automatically recognize hand-drawn graphical elements and transform them into elements that are part of a formal notation. This allows the designer to edit sketched interface designs and test the interaction of a sketched component or widget. But these tools did not show a lot of promise in everyday work, since the recognition of elements is still quite rudimentary, highly domain-specific and restricted to a given set of recognizable elements. Therefore, the support is not sufficient to convince users to switch from analog sketching to digital, tool-based sketching, although the approach seems appealing (Plimmer and Freeman [2007], Schmieder et al. [2009]).

Another toolset augments analog tools with digital capabilities.

Another, rather small set of tools that enhances the sketching experience and is not primarily concerned with automatic recognition of sketched elements or their beautification, is a recent field of interest. With these tools sketches stay sketches, but the analog tools get augmented with digital capabilities. One motivating factor behind these tools is to provide a suitable setting for idea generation by supporting and not omitting the Design Flow and applying the convenience of the digital world.

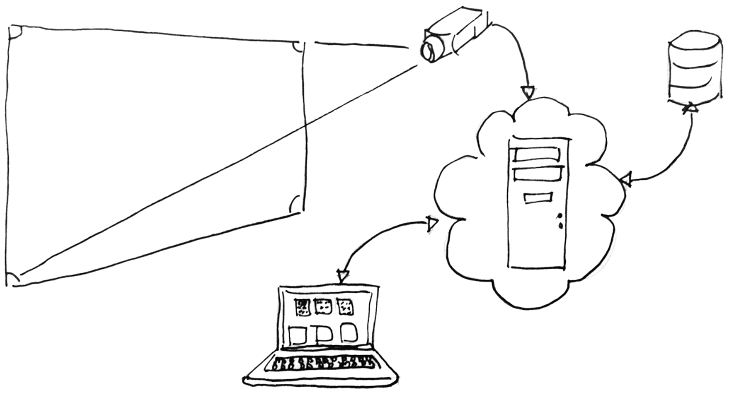


**Figure 2.3:** The Reboard system architecture introduced by Branham et al. [2010]. Sketches on the whiteboard are captured and retrievable via a calendar-like user interface.

ReBoard is a whiteboard capturing system that archives whiteboard content.

Branham et al. [2010] address the limited space of a whiteboard and the need for erasing sketches despite the fact that these ephemeral sketches may have a temporal value by introducing a system called ReBoard that automatically captures whiteboard images and archives them for later reference, so that the images can be accessed through a user interface with a calendar-like view. The progress and the results of collaborative and solo sessions get documented and are reusable in the future. For instance, a sketch drawn to help understanding a concept may be used later on as a starting point in a brainstorming session for related ideas. In order to not interrupt the Design Flow the system captures whiteboard images automatically by tracking changes on the whiteboard.

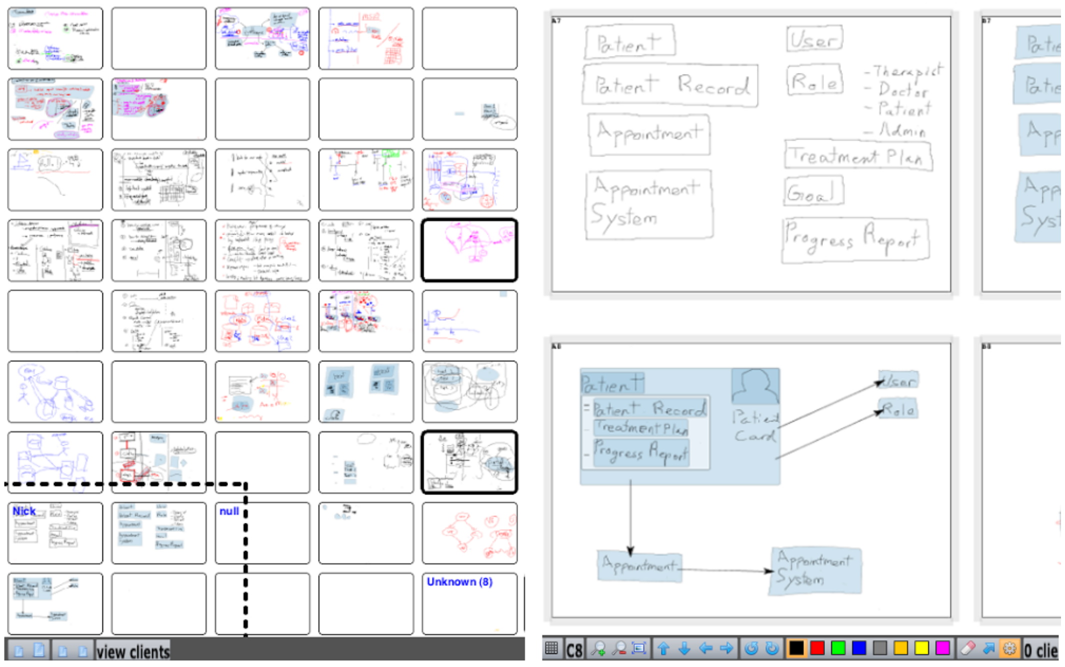


**Figure 2.4:** Calico introduced by Mangano et al. [2010] is an intuitive sketch-based design environment for touch-based devices: (a) Grid View to manage multiple Canvases, (b) advanced manipulation possible with scraps (blue elements)

Calico augments a whiteboard or tablet with digital interaction techniques.

Mangano et al. [2010] also try to enhance the software design process on electronic whiteboards and tablet devices. Their software called Calico adds certain features and capabilities to an electronic whiteboard or a tablet device. By introducing multiple virtual whiteboards arranged in a Grid, the limited space of whiteboards is addressed. The concept of scarps, which are grouped graphical elements, adds the ability to copy and paste certain parts of a sketch in order to reuse them and start exploring multiple variations of the same initial scarp without having to redraw them.

Sketches are redrawn multiple times if future use is anticipated.

Notwithstanding the above, the production quality of sketches changes when sketchers are conscious about the possibility that their doodles and scribbles might get reused in the future. This is regardless of whether sketches are produced digitally or in an analog way. Therefore, the Design Flow may get restricted since sketchers will start to redraw a sketch multiple times in order to get a cleaner version of the original sketch that will be recognizable in the future by others or self (Branham et al. [2010]).

The goal of tools presented in this section is to support software developers by enhancing their sketching experience. Therefore, these tools are very closely related to the traditional, analog way of sketching. In contrast, the following section presents some re-engineering tools. These tools visualize already existing code and omit the sketching part completely. They are still of interest for this thesis, since they show the influence and implications of visualizations combined with source code.

## 2.4 Visualization Tools for Software Developers

Source code visualization tools visualize only existing code.

A source code visualization is a graphical representation of source code that is created automatically by the development environment. Hand-drawn sketches and diagrams are code independent and can be used to visualize thoughts and ideas that are not in the source code itself, whereas automatically generated visualizations are code dependent and can only be representations of already existing code. However, they can create these visualizations without any assistance of the user and always depict the current status of the source code.

Source code visualizations might be helpful, but do not have to be.

Early contributions with regard to better understanding of code and graphical visualization of code were tools like SeeSoft (Eick et al. [1992]) and have become more and more evolved with recent tools like CodeGestalt (Kurtz [2011a]), Code Bubbles (Bragdon et al. [2010]), and Code Canvas (DeLine and Rowan [2010]). Corresponding user studies suggest that these tools can help in understanding code, but may also lead to an additional layer of confusion.

Code Canvas: Shifting focus from Code to Code Maps

In particular DeLine and Rowan [2010] give an interesting insight to the reasoning of their approach regarding the scenarios from subsection 2.2.1. Their re-engineering tool called Code Canvas is designed to support developers when trying to understand existing code, designing and refactoring existing code, and in ad-hoc meetings. At the center of their approach is the Code Map. The Code Map is a visual representation of a software project on a

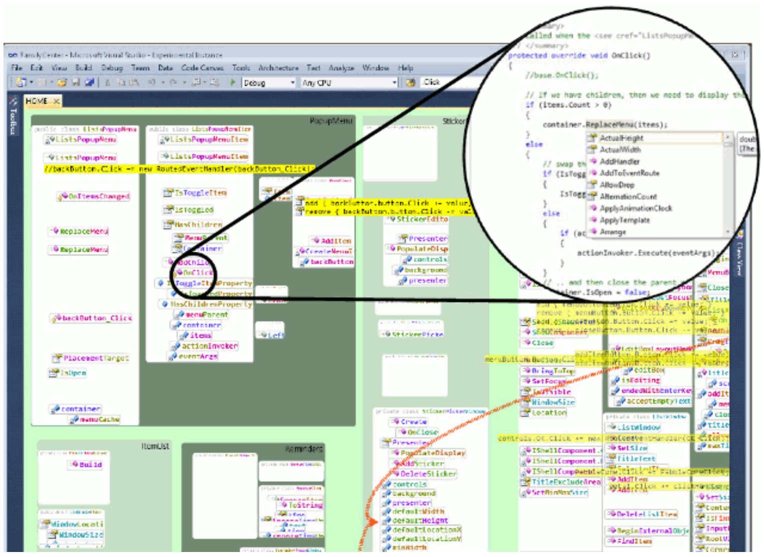


**Figure 2.5:** Code Canvas introduced by DeLine and Rowan [2010] is a Microsoft Visual Studio plug-in. It replaces the tabbed documents with a zoomable code map.

single map. Such a map should help developers to keep the mental model and give them better orientation while navigating through code. Starting with a paper-prototype (Cherubini et al. [2007a]), Code Canvas was implemented as a Microsoft Visual Studio plug-in. Although the paper-prototype was based on whiteboard drawings created by a team of developers, the resulting Code Map was a print out of stylized drawings created with a computer program. The software prototype is a zoomable and pann-able Code Map with the ability to hide certain details according to the zoom level in order to provide a microscopic and a macroscopic view of the project. In addition the user can open multiple canvases of the same Code Map at the same time to deal with different situations like working on a new feature on one Canvas and explaining some part of the project to a team member on a second Canvas. This enables the developer to simply return to the first Canvas and resume the initial task after the conversation with the team member ends and the second Canvas is no longer needed.

Code Maps are helpful to new team members.

In the case of Code Maps new team members to a project reported that they found visual representations helpful.

But still, re-engineering tools cannot visualize the mental model behind the code and that is something that is notably needed when trying to understand unfamiliar code.

## 2.5 Consequences for this thesis

The previous sections show the results of the literature review and provide a basic understanding of sketches, sketching in general, and sketching in the domain of software development. Moreover, some tools are introduced that support developers when sketching and some tools that provide visual representations of existing code. In summary, the following aspects of the literature review have influenced the development of this thesis:

Sketching supports ideation and communication.

Sketching is an established tool for ideation and communication. Sketches have the property of being sketchy, incomplete and ambiguous. Both the incompleteness and the ambiguity promote new thoughts and ideas. In terms of sketches being a means of communication, sketches can be used to enhance a conversation by offering an additional way to better understand the discussed issues and fill the gaps that spoken language may create.

Sketches are important when understanding, designing/refactoring, and in ad-hoc meetings.

Research focusing on sketches and software developers shows that software developers definitely create hand-drawn sketches and diagrams in various phases of a project and in different scenarios. Three scenarios were identified as the most important, when it comes to creating sketches: understanding, designing/refactoring and ad-hoc meetings. In all three scenarios software developers preferred hand-drawn sketches. Moreover, software developers revisit sketches and reuse them for multiple purposes like clarifying certain aspects for others or using sketches as a starting point in brainstorming sessions.

Sketches should stay sketches.

Computer-based sketching tools provide a poor recognition and, depending on the tool, sketches lose their ambiguity through that computerized recognition and following beautification. Moreover, sketching tools impede the Design Flow by distracting from the actual sketching task and redirecting the focus to rather unimportant aspects like colors, fonts and alignments.

Source code visualizations created by re-engineering tools seem to be promising in terms of understanding existing code, but user studies were not able to show that re-engineering tools provide a better understanding of existing, unfamiliar code.

Conclusions for this thesis

When explaining or conveying a mental model, sketches are a very useful and often applied tool. So it seems standing to reason to use created hand-drawn sketches and connect them to already existing source code to enhance the understanding and support the software comprehension process. One added value of sketches is the ability to capture planned functionalities and features.

That is why I believe that the connection of hand-drawn sketches and source code is a reasonable next step. The connection should merge the advantages of sketches and re-engineering tools and cancel out the disadvantages mentioned before.

# Chapter 3

# Initial Study

Gathering first-hand experience by visiting a company.

The literature review provided valuable insight into related work conducted in this particular field. However, in order to gather first-hand experience about the use of sketches during everyday work, I visited a company that provides IT solutions and consulting services to the energy industry located in Aachen, Germany.

Two meetings were observed and three informal interviews conducted.

The company employs around 300 people and created a revenue of 26 MM EUR in 2011. The IT department implemented the agile software development framework Scrum (Schwaber and Beedle [2001]) for their projects including daily standup meetings and weekly team meetings to expedite and plan the progress. I was able to sit in on two meetings of the software architects board and conducted three informal, unstructured interviews: one with the head of software architecture and two with software developers.

Derived from the data gathered during the meetings and the interviews, I present representative vignettes embedded in a narrative:

Co-workers interrupt each other to ask questions and gain knowledge about source code.

*Lea is the head of a software developer team and the head software architect. She has been working at the same company for 20 years and witnessed the releases of many products. That is why her team members call her the "dinosaur". At the moment her team is working on a new platform that will combine a set of smaller programs that have been implemented by different groups of the* *company. The company implemented an agile software development framework to manage this project. One of many tasks is to provide legacy support in order to be compatible with older versions. Since she was involved in all prior releases, Lea is the go-to person, when open questions arise and her team members have difficulties understanding the source code. So she constantly gets interrupted in her day-to-day work. In addition to all the scheduled team-meetings, she has many smaller, unscheduled, and informal meetings that happen in the hallway or before and after the main meetings.*

Sketches are used to provide a foundation for conversations and to support better understanding.

*During a scheduled team meeting some points have to be addressed concerning the software architecture of the new platform. All team members know the current architecture by heart and Lea can just talk about her concerns and suggestions: Everyone can follow her train of thought. Han has a suggestion to solve one of the issues and he presents a new idea he already discussed with Becca. They present their idea verbally. Luke, the newest team member, has problems following their mental leaps and is confused. He asks if someone could scribe the idea onto the whiteboard. Becca gets up and starts sketching (see Figure 3.1 1). She sketches some very sketchy large-scale elements to provide some orientation, so that everyone in the room can understand the context in relation to the software architecture. Then she starts to sketch in a more detailed way, while Han is repeating his idea. He matches his talking pace to the time Lea needs to sketch the spoken. Now Luke is able to follow the conversation, which is true now for some of the other developers, too. At the end of the meeting Lea takes a picture of the whiteboard with her mobile phone (see Figure 3.1 2). She sends it to Han and asks him to add the picture to the according task in the task tracking tool (see Figure 3.1 3).*

Sketches are created in ad-hoc meetings to explain source code design.

*Luke goes back to his desk and wants to resume the task he was assigned earlier, i.e., to fix a bug. He is trying to understand the code that is connected to the bug. He tries to reproduce the bug to get a hold of all components involved. He starts to make smaller changes, to see which influence these changes have. He takes a look at some visualizations that are provided by the development environment: he now understands some of the dependencies. But some information are still missing. He does not know how to change the code, since he cannot understand why the code is written the way it is. It makes no sense to him. Then he takes a look at the wiki that is used for internal documentation. But there is*

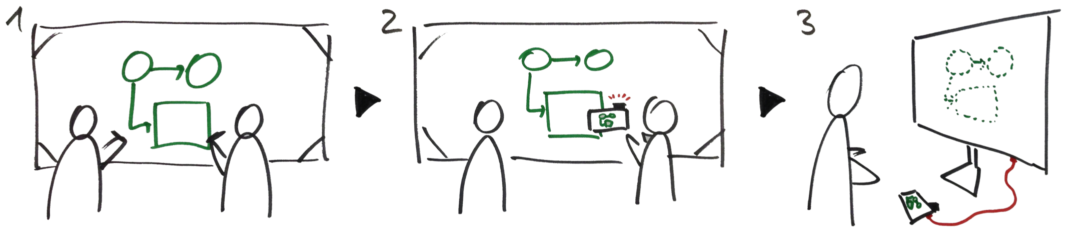


**Figure 3.1:** A sketch created on a whiteboard is digitalized and saved on the computer. Sketches are archived, but rarely used again. 1) collaborative creation of sketch on whiteboard, 2) digitalize the sketch with a digital camera or smartphone, and 3) upload the photo to the computer.

*just a lot of text and some code snippets of which he can find only few in the source code. The text is only partially helpful since the last change was made more than a year ago. He decides to go to Lea and ask her for help, because he was not able to grasp the concept behind the interaction of the involved components. Lea opens the according code in her development environment and start to explain. She clicks and jumps very quickly through the project. Then she takes a sheet of paper and starts to scribble some elements, connections, and dependencies. The explanation was a little fast, but Luke asks Lea if he could take the sketch with him in order to go through it on his own after returning to his desk. After some time Luke still cannot figure out how to fix the bug. So he takes the sketch to Han and asks him for help. Han takes a look at the sketch and adds some elements and lines. Now everything falls into place for Luke and he can return to his desk, where he successfully fixes the bug.*

Documentations are outdated and incomplete.

*Luke fixed his bug and starts fixing another bug. At first he tries to solve the problem by himself, but then he asks Lea for help eventually. Luke runs into Lea in the hallway, where whiteboards are mounted to the wall. Lea grabs a marker and sketches the rough conceptual structure and the reasoning behind the code that is relevant to fix the bug on the nearest whiteboard. Now Luke has all necessary information. He also has some ideas on how to improve the structure, which they explore by sketching additional elements. Since Luke cannot take along the whiteboard, he takes a picture with his smartphone and transfers that picture to his desktop computer in order to glance at it while fixing the bug.*

Sketches are digitalized and archived for future use.

*After leaving the hallway meeting, Lea realizes that some of the elements she sketched on the whiteboard were already depicted*

Sketches are helpful to new team members.

*on the first sketches created for Luke. She starts to ponder: Any other new team member might have similar problems to get on board and catch up with all the other team members. The visualizations of the IDE and the wiki were only partially helpful, since the code snippets in the documentation were outdated and there was no reference to the updated code segments. She has an idea to collect all the sketches and make the accessible for other new team members. But then she remembers: She tried to do something like that about five years ago for another project. She wanted to sketch and draw everything about the current project at that time, but after a month or so she surrendered, because the project progress was faster and she could not keep up.*

Sketches are hard to maintain.

The drawn conclusions from our gathered experiences are:

1. **Knowledge is in the head:** The knowledge about source code of a project is mainly in the heads of the software developers and is recorded very rarely and mostly for external documentation, i.e., documentation for clients. But an external documentation is not designed for mental models of the source code or the design rationale. Sketches are a adequate way to externalize the knowledge in the head about source code design.

2. **A tool of communication:** Sketches are created for self or in meetings (mainly between two and four people) in order to explain complex issues and to ensure a common knowledge base. The level of detail may vary in the same sketch if useful. Sketches support communication and provide an addition channel along with spoken language.

3. **Helpful tool for new team members:** New team members need support and assistance in order to catch up with the existent, but undocumented knowledge within a development team. They often interrupt co-workers and engage them in a conversation during which sketches are created for explanatory reasons frequently. These sketches are very valuable and helpful to the new team members.

# Chapter 4

# Fundamentals

An approach to communicate the source code design through sketches and, hereby, connect hand-drawn sketches with source code was not found in the literature review. Therefore, it is important for me to review the fundamentals involved, when talking about a connection between sketches and source code. These fundamentals provide a common ground on which the prototyping process was based on. This chapter provides an overview of these fundamentals and is structured as follows:

**Section 4.1—"Layouts and Views"** presents basic layout techniques of common software development environments as well as ways content is viewed. This section then introduces two views to integrate sketches into an integrated development environment (IDE).

**Section 4.2—"The Connection Points"** defines the termini that are used to create a connection between sketches and source code.

**Section 4.3—"The Connection between Connection Points"** introduces two ways to display a connection between Connection Points.

## 4.1 Layouts and Views

The basic views of an IDE as the starting point for exploration.

The following basic views are provided by common software development environments and provide a starting point for further exploration on how and where to integrate sketches:

- The **content area** displays the source code of a selected file.
- The **project tree** displays the hierarchical structure of the project using folders and filenames to support the navigation through a project.
- **Panels** provide additional tools and information.

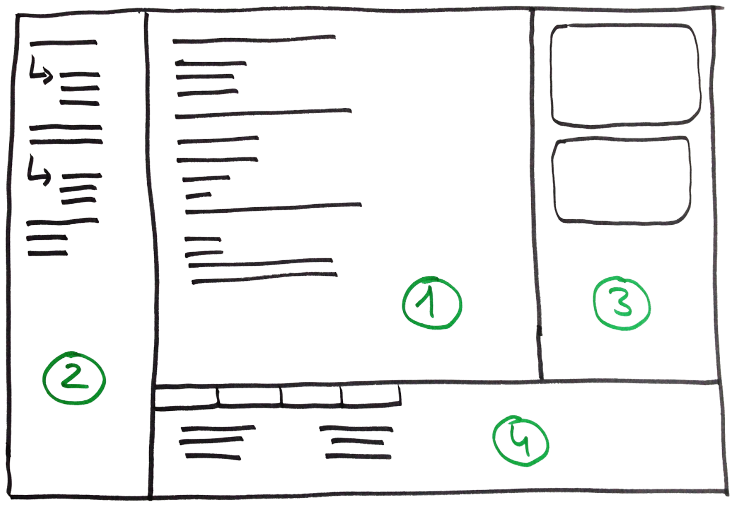


**Figure 4.1:** This sketch depicts basic areas of common IDEs: 1) content area, 2) project tree, 3) side panel and 4) bottom panel

Two basic layout techniques for views used in IDEs.

In order to add a view to the IDE, which is able to hold sketches, I present two layout techniques that emerged as fundamental ways to compose and arrange multiple views:

1. The **Split Layout** introduces an approach in which two or more views share a common space. A split can

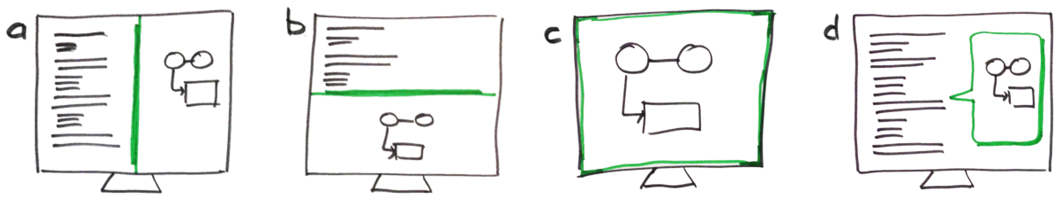


**Figure 4.2:** This sketch depicts examples of the basic layouts in which sketches are integrated into a software development environment: a) Split Layout (side panel), b) Split Layout (bottom panel), c) Overlay Layout (fullscreen) and d) Overlay Layout (floating)

be applied in two ways: horizontally (top-bottom) and vertically (left-right). Applying this layout recursively, a grid of views can be achieved, so that views like the content area, a side panel or a bottom panel can be displayed at the same time (see Figure 4.2 a, b).

2. The **Overlay Layout** implements a layer on top of another view. The visibility of that layer can be toggled, so that the Overlay View appears on top of the covered, underlying view or is not visible at all. This implies that it is not possible to work with the part of the underlying view that is covered and, therefore, hidden by the overlaying view. Different levels of transparency are a possible solution for this problem. Examples for this layout technique are floating windows and tabbed document interfaces (see Figure 4.2 c, d).

Two types of views to display and arrange sketches in IDEs:

Having introduced the basic layouts that can be used to place views in an IDE, the following two types of views support the display of sketches. Both differ in the way how value is attached to the placement and position of a sketch in relation to other sketches within the respective view.

The Map View respects the spatial arrangement of sketches.

1. The **Map View** provides a fixed position for all sketches in relation to each other and, thereby creates a structure as a whole, i.e., a map. Similar to having multiple elements in one sketch that have a relation to each other, grouping several sketches in a certain way spatially can have the same effect. Spatial memory can be supported by using the arrangement of

sketches like a map. Moreover, this view is independent of source code and the arrangement has a value in itself.

The Context View disregards the placement of sketches and allows various techniques to arrange sketches.

2. The **Context View** integrates sketches by displaying them on demand as an additional information about the corresponding source code. Access to the sketches is gained only via source code and not through the view itself. The display of contextual information about source code in the form of sketches can be realized in many different ways like panels or overlays. Spatial arrangements between several sketches are ignored and are highly dependent on predefined layout rules, e.g., tiles and slideshows, within the view.

## 4.2 The Connection Points

Connection Points are the termini of a connection.

In order to see how a connection between a sketch and source code can be realized, I explored the individual and elementary components and they are presented hereafter. According to the Oxford Dictionary a connection is defined as "a relationship in which a thing is linked or associated with something else". Set theoretically speaking the connection between sketches and source code is a binary relation between a set of sketches and a set of code. One element of the set of sketches can be connected to one or multiple elements of the code set, or vice versa. The following subsections will address the different kinds of elements of each set, which are the Connection Points.

### 4.2.1 The Connection Points of Source Code

Syntactical elements that can be used as Connection Points for source code.

A Connection Point in the source code is anything that can be selected by the user. Thus the following syntactical elements can serve as a Connection Point (see Figure 4.3):

- **single character** (atomic unit)
- **word** (sequence of characters)

- **line** (sequence of words)
- **block** (sequence of lines)
- **file** (sequence of blocks)

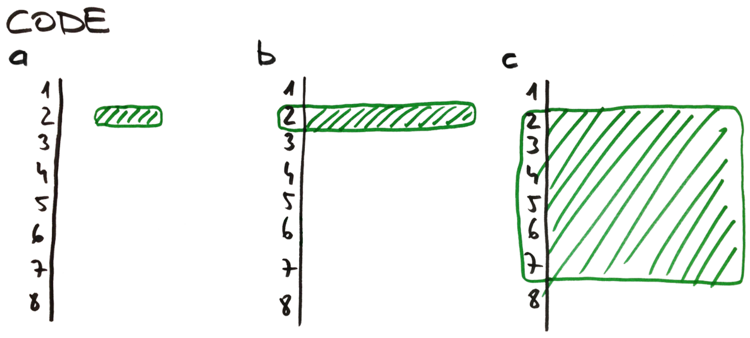


**Figure 4.3:** Examples of Connection Points in the source code: a) word, b) line, c) block

The above structure of Connection Points allows flexibility and a certain freedom: Semantical elements can be composed out of the syntactical elements listed above, e.g., a function or a method is a block and the declaration of a variable is a word or a line.

Semantical elements of Connection Points of source code.

### 4.2.2 The Connection Points of a Sketch

Assuming that a sketch is placed on a two-dimensional plane, the Connection Point of a sketch can syntactically be described as a geometrical element (see Figure 4.4). Such an element can be a

Syntactical elements that can be used as Connection Points for a sketch.

- **spatial point** (atomic unit),
- **line** (sequence of points) or
- **geometrical shape** (points and lines).

The semantical elements of a sketch can be composed out of syntactical elements: The Connection Point to a sketched circle-object can be any of the syntactical elements, i.e., a point, a circle, or any other shape.

Semantical elements of Connection Points of a sketch.

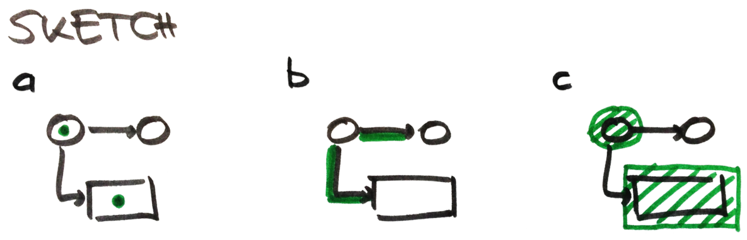


**Figure 4.4:** Examples of Connection Points on a sketch: a) point, b) line, c) geometrical shape

## 4.3 The Connection between Connection Points

A connection can be established between two or more Connection Points in a visualized or an indicated way.

The connection between source code and sketches links two or more Connection Points with each other and establishes a relationship between them. During initial ideation sessions to explore the possibilities many generated ideas were heavily dependent on the view that was chosen to integrate sketches into the IDE. Nevertheless, all ideas generated could be assigned to either of these two categories:

A visualized connection can be traced with the finger from one Connection Point to another.

- **visualized connections:** The most prominent example of a visualized connection of two or more Connection Points is a line between two elements. Using a line in order to connect elements is a common and natural way: The line symbolizes the path that has to be taken to get from one element to another element. Hence, the visualized connection is a connection that can be traced with the finger and therefore no mental effort is necessary in finding all elements that are in relation to each other.

An indicated connection adds characteristics to indicate a relation between Connection Points.

- **indicated connections:** A more subtle approach is indicating the relationship by adding unique characteristics to the Connection Points involved in the relation in order to differentiate them from other Connection Points that are not part of the relation. Examples for indicating techniques are color coding and different shapes like stars, triangles, dots or pins that group and, thereby, link certain elements together.

# Chapter 5

# Prototyping

The development of the prototype followed a cycle of design, implementation and analysis (DIA cycle) with users being involved in the process. The prototyping process started on paper and evolved into a software prototype, so that the connection of sketches and source code could be evaluated by software developers and programmers. This chapter describes the prototyping process and is structured as follows:

**Section 5.1—"Participants"** gives a short description of the participants, who were involved in the prototyping process.

**Section 5.2—"Methodology"** provides insight into how the prototyping process was carried out.

**Section 5.3—"Design decisions"** describes the design rationale of the prototyping process. Design decisions are presented for the choice of layouts and views as well as Connection Points and the connections between them.

**Section 5.4—"Implementation"** describes the results of the implementation phase of the software prototype by addressing its Look and Feel as well as its navigational behavior.

## 5.1 Participants

Four computer science students and two professional software developers provided continuous feedback during the analysis of the prototype.

During the development of the prototype, interviews, brainstormings, ideation and feedback sessions were conducted with six people in all, i.e., mainly one or three in each iteration. Four participants of these sessions were undergraduate students of computer science and two were professional software developers. Each of the students had experience with software development, whereas two worked in team projects and two worked on solo projects at that time. One of the two participating software developers was a freelancer and the other one was an employee at an IT company.

## 5.2 Methodology

The prototype was developed using the DIA cycle.

Following the rationale of the DIA cycle, ideas were generated during early brainstorming sessions to get a rough insight on how the connection of sketches and source code could be realized and what the interaction should look like. The more evolved the prototypes got the more detailed the implementation became. Design decisions were made and implemented during the process. Once new ideas and changes were mentioned during the analysis of a software prototype feature, these were again explored on paper and implemented hereafter. Smaller adjustments and fine tunings at the end of the development process were then implemented directly (see Figure 5.1).

## 5.3 Design decisions

During the prototyping process approaches were explored and design decisions were made.

In the course of prototyping many possible configurations and variations of layouts, views, Connection Points and connections types were explored and discussed. While getting input from participants involved in the process and considering findings of the literature review, certain design decisions I made, that led to the software prototype. In the

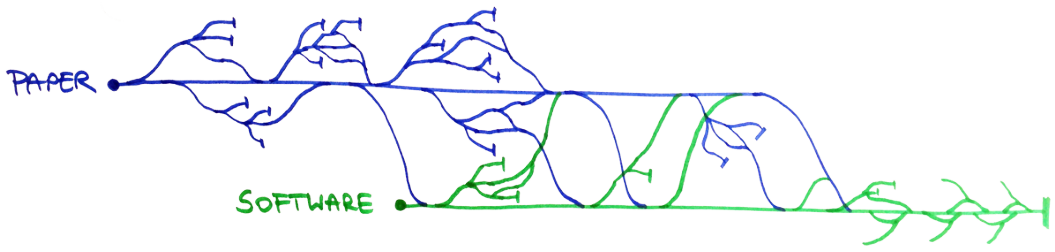


**Figure 5.1:** The prototype development process started with paper prototyping and evolved to a software prototype. The figure depicts the process figuratively. Parting lines represent the generation of new ideas. During the analysis of these ideas, design decisions were made that influenced the prototype represented by merging lines.

following subsections my decisions and their rationale are presented.

### 5.3.1 Layouts and Views

The way a sketch is arranged in relation to the source code was a very vivid topic for the participants. Using sketches as a map with spatial arrangements and recognizable landmarks was favored over approaches in which the spatial memory was ignored.

Layouts and views were examined in ideation sessions.

1. The **Split Layout:** Participants noted that they would like to see the source code and the sketches at the same time, e.g., side-by-side. Most participants imagined a side panel on the right side of the source code since source code starts at the left edge of the content area, whereas the right part of the content area is mostly unused when common coding conventions are followed, especially, on current monitors with a 16:9 display ratio. Participants imagined a white canvas on the right side of the content area and liked the idea of placing sketches next to the corresponding source code. That is why the vertical Split Layout was one of the favorite layouts. A horizontal Split Layout was discussed as well, but was not liked at all: Loosing precious height in the content area, i.e., the amount of concurrently visible lines of code, was

The Split View allows for showing source code and sketches side-by-side. A vertical split was preferred.

seen as a clear disadvantage.

The Overlay View displays a view on top of another view and covers that underlying view.

2. The **Overlay Layout:** The mostly referenced variation of this layout was a fullscreen Overlay. Despite the fact that in this layout it is not possible to see the source code and the sketches at the same time, this layout was the participants favorite in terms of project navigation. The fullscreen Overlay Layout was compared to a map that should be helpful with regards to orientation within a project due to the strong use of spatial memory if the view allowed for fixed placements of sketches. This advantage was derived from experience with, e.g., computer games in which opaque overlaying maps provide overview and orientation.

Decision: Use vertical Split Layout and fullscreen Overlay Layout.

After talking to participants about the layouts, I decided to implement a vertical Split Layout and a fullscreen Overlay Layout. Participants were also consulted on how to display the sketches within a view:

Exploring the use for the Map View showed promise with providing orientation due to use of spatial memory.

1. The **Map View:** The possibility to give meaning to a position on a canvas by placing a sketch at that position, was requested by many participants. They talked about displaying the architectural structure of software by arranging the sketches in a certain way and thereby having a way of applying value to the arrangement and recording knowledge acquired while creating the sketches. This approach of visualizing sketches was compared to a whiteboard and its experienced ability to provide overview and orientation, since a fixed placement of sketches supports spatial memory.

The Context View was disliked, since it ignores the meaning of the placement of sketches.

2. The **Context View:** This particular view generated ideas that were very dependent on the way how the connection between source code and sketches was realized. One example mentioned, were tooltip-like floating bubbles. These bubbles would overlay the source code, but not cover it as a whole. Another idea for a Context View was showing all sketches, that were important in the specific context, in a filmstrip-like fashion integrated in a Split View. But, the Context View was not liked by the participants due to the

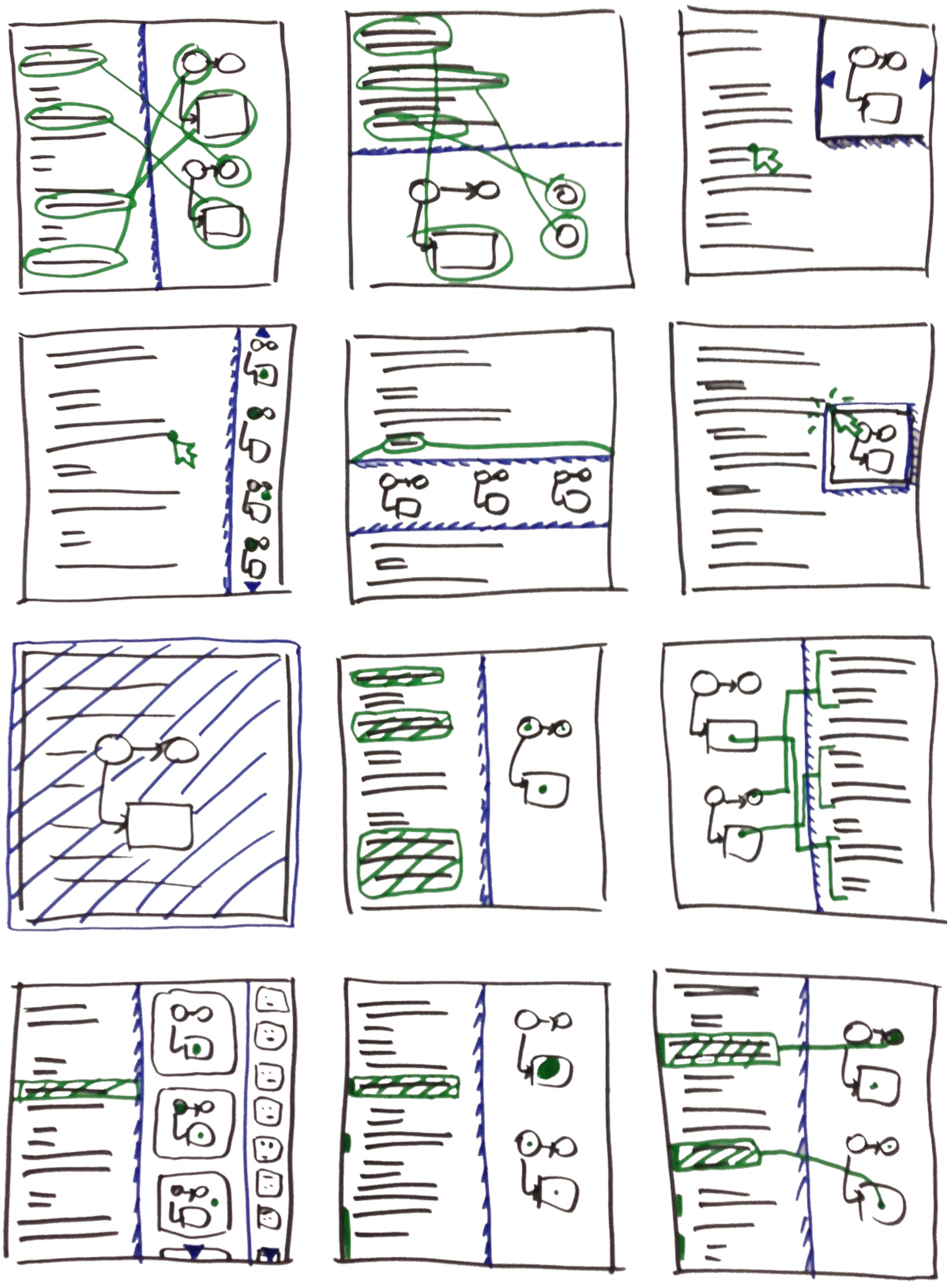


**Figure 5.2:** Results of an ideation session about the integration of sketches into the IDE. Overlay and Split Layouts determine the layout of Map and Context Views in relation to the source code. Visualized and indicated connections are used to link Connections Points to each other.

lacking use of spatial arrangements. Participants reported to have no use for sketches that are technically part of a whole structure, but are shown individually and detached from that structure.

Decision: Use the Map View.

With regard to these comments, I decided to not use the Context View at all and instead all views should have properties of the Map View to support spatial memory.

### 5.3.2 The Connection Points

Feedback showed, that participants want to use all syntactical elements as Connection Points for source code.

When talking about Connection Points with the participants, there was a consensus that they wanted to be able to use any of the syntactical elements (single character, word, line, block, file) of the source code as Connections Points. The reason was that they want to be able to select anything in the source code and connect it to a sketch without restrictions.

Participants were undecided on which syntactical elements to use as Connection Points of sketches: spatial points and shapes were favored.

In contrast participants were not very agreed on the Connection Points of sketches. Some participants were satisfied with spatial points and compared them to thumbtacks or fridge magnets that would suffice. Other participants wanted to be able to use all syntactical elements, e.g., they wanted to use a rectangular shape that would overlap a rectangle on the sketch. They liked the freedom of choice and the ability to use a plane as a Connection Point. However, one participant made a compelling argument against using geometrical shapes as Connection Points: Since a sketch already made use of geometrical shapes in order to create elements and objects and relations, the use of geometrical shapes as Connection Points would render many parts of the sketch redundant. In a way, the sketch would get replicated.

Decision: Use all syntactical elements as Connection Points for source code and use only spatial points as Connection Points for sketches.

In order to keep the whole process of connecting sketches with source code as simple as possible, the decision was made to implement all syntactical elements as Connection Points of source code, but only spatial points as Connection Points of sketches in the software prototype. Since the idea behind the software prototype is to provide the ability to connect sketches and source code with each other, the focus is exactly that, since introducing multiple and resizable shapes as well as different colors might shift the focus to refining and beautifying the Connection Points (Wong [1992], Goel [1995]).

### 5.3.3 The Connection between Connection Points

Visualized connections lead to clutter and indicated connections demand clear feedback to reduce mental effort.

Once a decision about Connection Points for both sketches and source code was made, the connection between Connection Points had to be established. A line between two elements is a commonly used way to show a relation between these elements. A visualized connection can instantly be perceived and all visible, involved elements can be found at a glance. But this was also mentioned as a drawback, since too many lines between multiple elements can lead to confusion and an overload of information. In contrast, indicated connections have no problem with clutter, though a mental component might be necessary, since the connection has to be made mentally, e.g., by recognizing and differentiating unique characteristics of related Connection Points of a sketch.

The first approaches sketched by me and participants showed outlined source code, outlined sketches and a line in between. At first participants thought about drawing the connection line directly on the source code and tested it in an early software prototype. Since there was no computerized management of those lines, this approach was quickly dismissed and participants asked for a cleaner and more subtle way.

No approach was found to combine visualized connections with a fullscreen Overlay Layout.

Another problem with regards to visualized connections was a fullscreen Overlay Layout: This layout provides a layer on top of the content area and visualized connections between layers of depth could not be imagined in any way. In this situation participants drew small dots on the paper prototype of the fullscreen Overlay View to indicated a connection to a sketch. These dots were then implemented in the software prototype for all connections between sketches and source code and participants responded well to these Connection Dots, which reminded them of fridge magnets or portals. Connection Dots do not create any clutter and do not replicate the sketch itself.

Participants preferred marked line numbers to indicated connected code segments.

Furthermore, participants did not like that code segments were outlined all the time, i.e., being marked. They preferred to indicated the connection by marking the corresponding line number in order to prevent overstimulation caused by multiple marked code segments.

Decision: Use Connection Dots and marked line numbers to indicated connections.

Hence, the indicated connection was persuade during the development of the software prototype and the Connection Dots are visual representations of Connection Points of a sketch that are actively involved in a connection between sketches and source code. On the side of the source, I decided to mark the line numbers of lines that were part of a connection.

## 5.4 Implementation

### 5.4.1 Platform

The prototype was implemented as an extension for Adobe Brackets.

I implemented the software prototype as an extension for the open-source code editor Adobe Brackets[1]. Adobe Brackets is a community-driven project and is built on top of web technologies such as HTML, CSS and JavaScript. It is still in its early development stages and is developed in the manner of the agile software development framework Scrum. During the course of this thesis Sprints 15 to 24 were released with major changes and feature additions.

Adobe Brackets provides a single, continuous selection metaphor.

The underlying core text editor of Adobe Brackets is CodeMirror[2] to which Adobe Brackets offers a very basic API, so that direct access to instances of CodeMirror is necessary and encouraged by the community. CodeMirror implements a single, continuous selection metaphor, meaning that there can only be one continuous selection at any given time. Hence, it is not possible to select multiple code segments that are intermitted by other unselected code.

[1]http://www.brackets.io
[2]http://codemirror.net

Known gestures like holding the shift key still result in a continuous selection with one starting point and one end point of that selection.

The following libraries were also used in order to implement the functionality of a connection between source code and sketches:

Additional libraries were used to implement the design decisions.

- **Sketch.js**[3] is a jQuery plugin that provides the basic functionality of sketching on an HTML canvas element. The plugin was extended with an undo function. It is used to support free-hand sketching.
- **Kinetic.js**[4] is an HTML5 Canvas JavaScript framework that enables high performance animations, transitions, node nesting, layering, filtering, caching, and event handling, among others. It is used to support managing sketches and their Connection Points.
- **JSON2.js**[5] provides a light-weight, language independent, data interchange format. It is used to save and restore all data about connections in an xml-file.
- **jQuery UI**[6] provides a set of user interface interactions, effects, widgets, and themes. This library is used for transitions.

### 5.4.2 The Look & Feel

According to the Design decisions made about the views that should be supported by the prototype, two views emerged as approaches to enable the connection of sketches with source code. During the implementation process, small feedback loops with participants helped defining and creating a look and feel for the connections between sketches and source code. In order to provide a holistic approach to the interaction, the following views present a similar behavior in terms of creating, managing and using connections between sketches and source code.

A similar look and feel was implemented for all views to provide a consistent user experience.

[3]http://intridea.github.io/sketch.js/
[4]http://kineticjs.com/
[5]https://github.com/douglascrockford/JSON-js
[6]http://jqueryui.com/

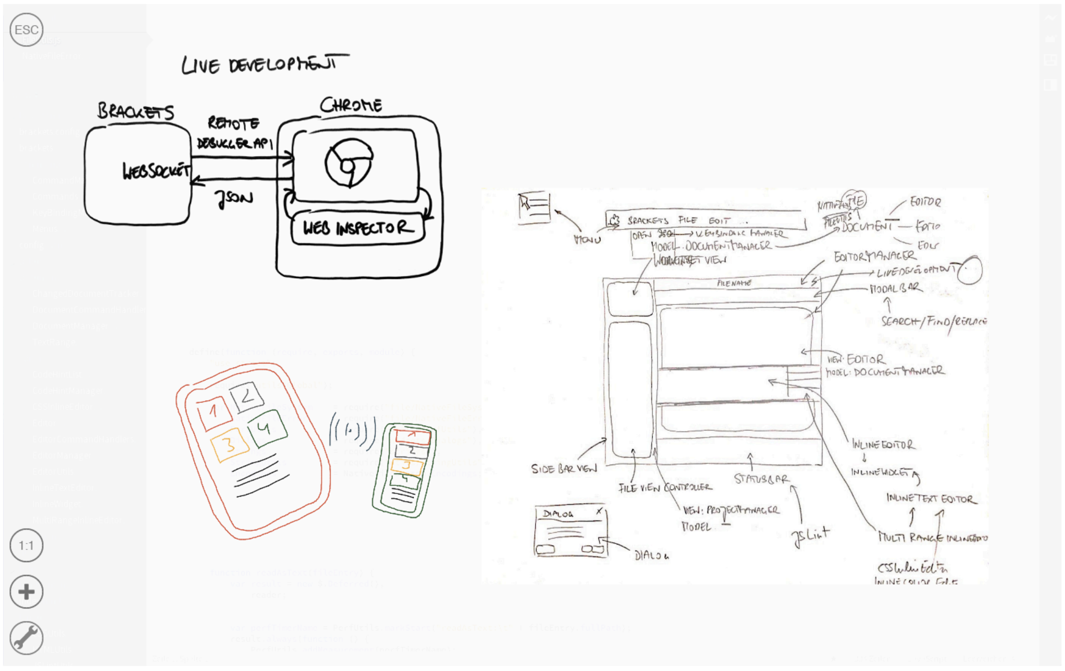


**Figure 5.3:** A screenshot of the Mission Control View: a semi-transparent, fullscreen overlay on top of the source code offers an overview of a project.

### The Mission Control View

The Mission Control View is a Map View integrated into a fullscreen Overlay Layout.

The Mission Control View implements a Map View integrated into a semi-transparent, fullscreen Overlay Layout with indicated connections. This view is a canvas of infinite size with a basic zoomable interface as the navigational metaphor in order to adapt to different screen size. The user interface elements of this view are integrated into the view itself, since the Mission Control View overlaps all other areas of the editor. Since this view is designed to provide an overview of the whole project, only one Mission Control View exists per project.

The Mission Control View provides two types of Connection Dots:

Moreover, the Mission Control View introduces two kinds of Connection Dots, which differ from each other in color, so that users can distinguish between the two types of connected Connection Points:

Code Dots are connected to code segments.

- **Code Dots** are dots that are connected to characters, words, lines and blocks of source code. A Code Dot has a visible counterpart at the further end of the connection, i.e., in the code. To indicate the corresponding connected Connection Point in the source code the corresponding line numbers are color-coded.

- **File Dots** are dots that are connected to a file within the project and have no visible counterpart on the source code side, since they are connected to the whole file.

File Dots are connected to files.

In order to create, manage and use the Connection Dots, the Mission Control View introduces two modes:

- **normal mode:** In this mode the user can use the Dots as a navigational and orientational tool.
- **edit mode:** In this mode the user can add, move and delete sketches as well as Connection Dots.

### The Sketchbar View

The Sketchbar View is the right frame of a vertical Split Layout, whereas the source code is in the left frame. This side-by-side approach allows for displaying sketches and source code at the same time. The user interface is integrated into the Sidebar that it provided by a default installation of Adobe Brackets. One Sketchbar View exists per file, so that each file can have its own sketches displayed alongside the source code.

The Sketchbar View is the right frame within a vertical Split Layout next to the source code.

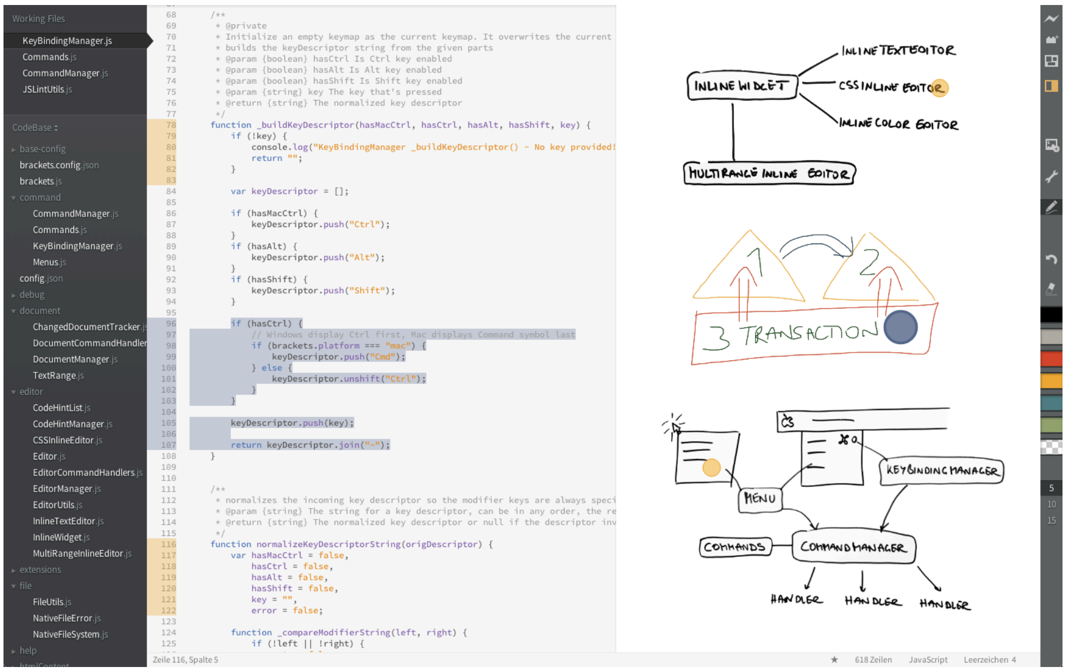
The Sketchbar View implemented in the software prototype uses only Code Dots due to the one-to-one relation of a file and a Sketchbar View. Therefore, File Dots are not used in this view at all.

Only Code Dots are implemented.

In addition to the normal and the edit mode which are similar to those implemented in the Mission Control View, the Sketchbar View introduces a third mode:

The sketching mode allows the user create free-hand drawings directly on the Sketchbar View.

- **sketching mode:** In this mode the user can sketch on the canvas provided by the Sketchbar View in order to quickly annotate sketches or make small additions.



```
    /**
     * @private
     * Initialize an empty keymap as the current keymap. It overwrites the current
     * builds the keyDescriptor string from the given parts
     * @param {boolean} hasCtrl Is Ctrl key enabled
     * @param {boolean} hasAlt Is Alt key enabled
     * @param {boolean} hasShift Is Shift key enabled
     * @param {string} key The key that's pressed
     * @return {string} The normalized key descriptor
     */
    function _buildKeyDescriptor(hasMacCtrl, hasCtrl, hasAlt, hasShift, key) {
        if (!key) {
            console.log("KeyBindingManager _buildKeyDescriptor() - No key provided!
            return "";
        }

        var keyDescriptor = [];

        if (hasMacCtrl) {
            keyDescriptor.push("Ctrl");
        }
        if (hasAlt) {
            keyDescriptor.push("Alt");
        }
        if (hasShift) {
            keyDescriptor.push("Shift");
        }

        if (hasCtrl) {
            // Windows display Ctrl first, Mac displays Command symbol last
            if (brackets.platform === "mac") {
                keyDescriptor.push("Cmd");
            } else {
                keyDescriptor.unshift("Ctrl");
            }
        }

        keyDescriptor.push(key);

        return keyDescriptor.join("-");
    }


    /**
     * normalizes the incoming key descriptor so the modifier keys are always speci
     * @param {string} The string for a key descriptor, can be in any order, the re
     * @return {string} The normalized key descriptor or null if the descriptor inv
     */
    function normalizeKeyDescriptorString(origDescriptor) {
        var hasMacCtrl = false,
            hasCtrl = false,
            hasAlt = false,
            hasShift = false,
            key = "",
            error = false;

        function _compareModifierString(left, right) {
            if (!left || !right) {
```

**Figure 5.4:** A screenshot of the Sketchbar View: a Split View that offers a side-by-side view on sketches and source code. Yellow line numbers and yellow dots on sketches indicate connections. The blue dot and the blue marked source code illustrate a selected connection.

### 5.4.3 The navigational behavior

Having the sketches and the source code side-by-side the navigational behavior has to be specified to support communication of source code designs.

After implementing the two views and the functionality of connecting the Connection Dots to Connections Points of the source code, the navigational behavior was very important. The premise was to provide an intuitive and supportive way in which a user interacts with the two views and the communication of source code designs is supported by showing requested information in a helpful way.

#### The Mission Control View

Participants asked to be guided from a sketch to the source code instead of jumping there.

This view allows the user to navigate through the source code by omitting the project tree and its folder structure. Participants wanted to be guided from a sketch to the corresponding source code segment in order to preserve some kind of orientation within the source code rather than jump from one to the other without any visual feedback and relying on external reference points like the position of the scrollbar to check if the position within a file changed, or checking the filename to see if possibly even the file itself changed.

Clicking on a Connection Dot will initiate a transition from the Mission Control View to the content area providing visual feedback for the user.

Therefore, clicking on a Connection Dot will highlight it by enlarging it and dissolve the Mission Control View gradually at the same time, so that the user gets visual feedback in reaction to the click and the source code is the focus again. If the clicked Connection Dot is connected to a code segment in a file different then the one currently opened, the corresponding file is opened. If, in addition, the clicked Connection Dot is a Code Dot the content area is scrolled to the connected code segment, i.e., the Connection Point of source code, which then is highlighted via a color-coded, permanent selection. The permanent selection of the connected code segment is active for as long as the user needs it and can only be unselected manually by clicking on the corresponding line numbers, which toggles the permanent selection as well as the highlighting of the corresponding Connection Dot on the Mission Control View. This way the user is provided with as little as possible, but as much as necessary visual feedback about the transition from sketch to source code.

Back-links can be used to orientate self within the project with the help of sketches.

Moreover, this view allows users to orientate themselves using **back-links**, i.e., using the connection between a sketch and source code starting from the Connection Point of the source code and following it to the Connection Point of the sketch. Participants testing the software prototype reported that they would like to see where certain connected code segments are located within the overall structure of a project. Clicking on a color-coded line number results in a highlighted Connection Point of the source code and shows the permanently selected code segment that is part of the connection; at the same time the corresponding Connection Dot is highlighted, i.e., enlarged, on the Mission Control View. In order to see and find the highlighted Connection Dot within the sketches, users have to toggle the Mission Control View.

A File Dot is highlighted automatically.

However, File Dots are highlighted automatically as soon as the corresponding file is opened in the content area, so that the user can toggle the Mission Control View at any time to gain orientation in the project with the help of highlighted File Dots and perceive the context in which the file is in relation to the rest of the project.

### The Sketchbar View

The navigational behavior is tied to the scrolling behavior of the views in a Split Layout.

Since this view allows the user to see both the source code and additional information about the source code in the form of sketches at the same time, the navigational behavior was much discussed, especially the scrolling behavior of both views, i.e., the content area containing the source code and the Sketchbar View, in relation to each other. Therefore, I explored the behavior on paper and implemented afterwards in order to proof the concept.

Both the content area and the Sketchbar View have the same total height to support placing of sketches in proximity to corresponding source code.

Participants wanted to place the sketches in relation to the source code. Hence, a sketch that is related to a certain part of the source code should be visible when the related source code is visible. But participants did not want to have a Context View, since they considered the support of spatial memory as essential. Participants suggested a Map View linked to the content area: A canvas with the same height as the total height of the file, i.e., the height of both the on-screen and off-screen part. So if a file consists of 1000 lines of code and a line has a height of 15px, then the total height of that file would be 15000px, as would have the Map View. With regards to the total height of an empty file or a file, that has a height lower than the editor window, I made the design decision to initialize the height of the canvas of Sketchbar View with the height of the Sketchbar View itself, which is dependent on the height of the editor window. The decision is based on the participants' comments that sketching often is done before the implementation of code is initiated, hence, the user should be able to import a sketch, even if no line of code has been written, yet.

Using intelligent synchronized scrolling for both views.

Moreover, participants suggested **synchronized scrolling** of the content area and the Sketchbar View, since they did not want to scroll both the content area and the Sketchbar View separately. After participants tested the implementa-

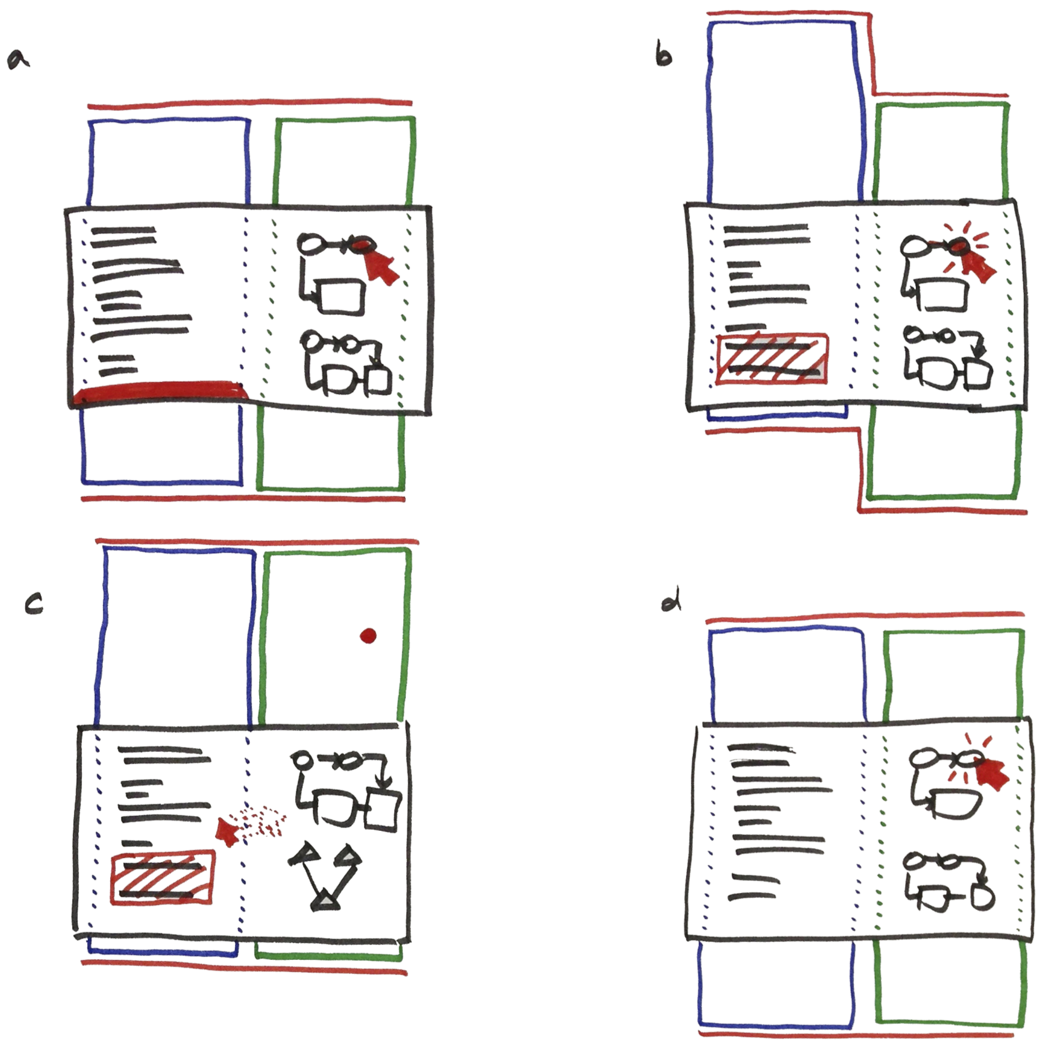
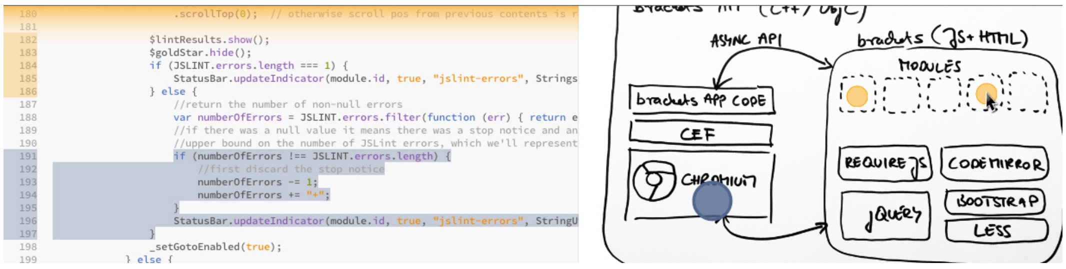


**Figure 5.5:** The scrolling behavior implemented in the Sketchbar View: a) hovering a Connection Dot indicates off-screen connected code segment, b) clicking a Connection Dot scrolls the code segment on-screen, c) entering the content area activates synchronous scrolling and syncs the two views, d) clicking on an active Connection Dot without having entered the content area before synchronizes the views.

tion of synchronized scrolling, they were irritated with the behavior when only one Connection Point of a connection was visible and the other one was off-screen. This is mainly the case, when a sketch on the canvas of the Sketchbar View represents multiple code segments in the source code and the corresponding code snippets are interspersed throughout the file. Therefore, a meaningful behavior of the inter-



```
                    .scrollTop(0);  // otherwise scroll pos from previous contents is r

                $lintResults.show();
                $goldStar.hide();
                if (JSLINT.errors.length === 1) {
                    StatusBar.updateIndicator(module.id, true, "jslint-errors", Strings
                } else {
                    //return the number of non-null errors
                    var numberOfErrors = JSLINT.errors.filter(function (err) { return e
                    //if there was a null value it means there was a stop notice and an
                    //upper bound on the number of JSLint errors, which we'll represent
                    if (numberOfErrors !== JSLINT.errors.length) {
                        //first discard the stop notice
                        numberOfErrors -= 1;
                        numberOfErrors += "+";
                    }
                    StatusBar.updateIndicator(module.id, true, "jslint-errors", StringU
                }
                _setGotoEnabled(true);
            } else {
```

**Figure 5.6:** A screenshot of the Sketchbar View: A yellow Connection Dot and a yellow line numbers indicate a connection. The highlighted blue Connection Dot and the blue marked code segment depict an active, selected connection. Hovering over the right, yellow Connection Dot, the top edge is highlighted, which indicates that the connected code segment is off-screen.

action with connections had to be defined with regards to this particular view, because both are tied closely together.

Feedback needs to be provided when one Connection Point of a connection is off-screen.

If parts of a connection are not visible, be it due to the virtual limitations of the editor window or due to the real limitations like the size of a display, the user still needs to get meaningful feedback on what the further end of the connection is. I implemented the following behavior:

Hovering over a Connection Dot marks the corresponding code segment for the duration of the hover.

- **Hovering a Connection Dot** with the mouse pointer highlights the Connection Dot itself and marks the corresponding code segment if it is on-screen. However, if the corresponding code segment is off-screen its position is indicated either at the bottom edge of the content area if the code segment is located further down in the source code or at the top edge in the opposite case (see Figure 5.6: yellow gradient at top edge). Upon leaving a hovered Connection Dot with the mouse pointer it gets unhighlighted and the corresponding code segment is unmarked. Indicators are hidden if the corresponding code was off-screen.

Clicking on a Connection Dot turns the code segment mark into a permanent mark.

- **Clicking a Connection Dot** sets both the highlighting of the Connection Dot and the marked code segment, which was already marked during hovering the Connection Dot, as permanent (see Figure 5.6). If the corresponding code segment is off-screen the content area scrolls up or down until it is on-screen (see Figure 5.5b); in this case synchronized scrolling is deactivated, with an offset between the two views of the Split Layout being the result. Upon leaving a clicked and highlighted Connection Dot nothing happens. The connection is highlighted and stays highlighted until it is manually unhighlighted by clicking on the Connection Dot again (see Figure 5.5d) or by clicking on the line numbers of the corresponding code segment. The offset between both views is eliminated and the synchronized scrolling is reinstated as soon as either the mouse cursor enters the content area (see Figure 5.5c) or the Sketchbar View is scrolled.

The content area is asynchronously scrolled to the appropriate line if the connected code segment is off-screen.

Similar to the Mission Control View, back-links are available for this view and behave in the same way: A click on the line number marks the Connection Point of the source code, i.e., the connected code segment, and the corresponding Connection Dot is highlighted in the Sketchbar View. If the Connection Dot is off-screen, synchronized scrolling is deactivated and the Sketchbar View is automatically scrolled so that the highlighted Connection Dot is on-screen. Synchronized Scrolling is reinstated if the line number is clicked so that the code segment is unmarked and the Connection Dot is unhighlights if the content area is scrolled or if the Sketchbar View is entered with the mouse cursor.

Back-links enforce asynchronous scrolling if the corresponding Connection Dot is off-screen.

The implementation of such a behavior is highly dependent on and was only possible due to the single, continuous selection metaphor of the editor. It is not possible if a connection consists of more than two Connection Points and more than one is off-screen.

This scrolling behavior is dependent on the single, continuous selection metaphor.

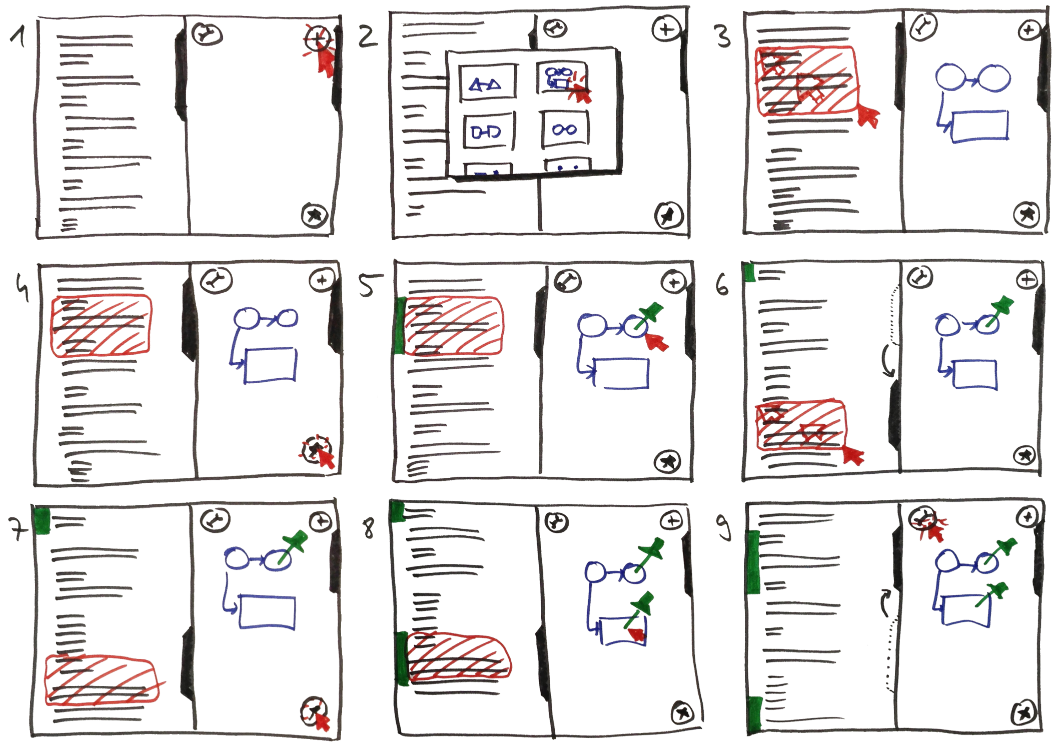


**Figure 5.7:** A use case of the Sketchbar View depicting the creation of connections and its scrolling behavior: Add a picture to the Sketchbar View (1) by selecting it for the native file system (2). Select a code segment that shall be connected to the added sketch (3). Add (4) and place (5) the Connection Dot on the sketch. Since the edit mode was entered upon adding a picture, asynchronous scrolling is activated. Scroll the content area and select another code segment that shall be connected to the sketch (6). Again add (7) and place (8) a new Connection Dot for that connection. Leave the edit mode (9), so that synchronized scrolling is activated and the content area is automatically scrolled to the same position as the Sketchbar View is currently.

# Chapter 6

# Evaluation

To test how the software prototype works for practitioners, I conducted a user study at RWTH Aachen University in which I explored five hypotheses. This chapter is structured as follows:

**Section 6.1—"Experimental Setup"** presents the setup of the between groups user study by explaining the recruitment process of participants and introducing the two tasks, participants had to complete. Moreover, the two conditions of the study are described in detail as is the methodology applied during sessions.

**Section 6.2—"Results"** presents some general information about the participants joining the user study as well as the results of the sessions and their influence on the five hypotheses. The chapter ends with a compilation of insightful, interesting and distinctive comments made by the participants.

**Section 6.3—"Discussion"** briefly discusses the results and observations of the user study.

In order to investigate if the connection of sketches and source code can communicate the source code design and, therefore, be an additional source of information that helps software developers and programmers in their software comprehension process, I suggest and explore the following five hypothesis:

**H1** Given a time-constrained task that requires browsing and understanding source code, **more programmers can solve the tasks correctly using the software prototype with a sketch-enriched code base** than using a default installation of the editor Adobe Brackets with a regular code base and having the sketches presented on paper.

**H2** Using the software prototype with a sketch-enriched code base, **programmers can solve tasks that require browsing and understanding source code more quickly** than using a default installation of the editor Adobe Brackets with a regular code base and having the sketches presented on paper.

**H3** Working on a sketch-enriched code base and using the software prototype during tasks that require browsing and understanding source code, **programmers look at sketches more often** than when using a default installation of the editor Adobe Brackets and having the sketches presented on paper.

**H4** Working on a sketch-enriched code base and using the software prototype during tasks that require browsing and understanding source code, **programmers look at sketches longer** than when using a default installation of the editor Adobe Brackets and having the sketches presented on paper.

**H5** Programmers (subjectively) find that the connection between sketches and source code is an additional tool, that supports their software comprehension process by helping them to **understand the mental model behind the code**.

## 6.1 Experimental Setup

Hypothesis H1-H4 can be tested by performing quantitative measurements. Supporting H5 requires qualitative methods, such as a semi-structured post-session interview and observing participants working on tasks. In this section we describe the way we recruited participants and present in detail the setup of the experiment that was used during the user study in order to explore the five hypotheses.

### 6.1.1 Participants

Invitations were sent to computer science students and professional software developers.

In order to evaluate the software prototype graduate and undergraduate computer science students as well as alumni from RWTH Aachen University were invited via an email circular to participate in the user study. Several days before the user study was closed all participants who received the initial circular and had not yet responded, received individualized reminder emails. In addition to that also professional software developers were directly approached and invited via email to participate in the study. All participants were compensated by entry in a drawing for a 50 EUR gift certificate.

### 6.1.2 Tasks and Conditions

Participants need to understand the source code to successfully complete two tasks.

The evaluation of the software prototype was designed in such a way that Hypothesis H1-H4 could be tested with quantitative measures. We chose two tasks, which should be completed by the participants. Both tasks required the participants to read and understand the source code in order to successfully complete them. The focus of both tasks was on the navigation between methods and files within a project, rather than creating or changing existing code, since finding the right locations in the source code and using a reasonable path to get there, should indicate basic comprehension of the source code design and its mental model.

Task 1 is fixing a bug.

**Task 1** is fixing a bug, that was issued within the Adobe Brackets community after the release of Sprint 19 and was fixed with Sprint 20. The Brackets community identified the issue[1] suitable for beginners task and therefore it is appropriate for getting used to the editor and the code base. In order to fix the bug a certain part of a regular expression within an if-statement needs to be removed. I limited the working time for this task to 20 minutes.

Task 2 is adding a menu item and the corresponding functionality.

**Task 2** is adding a new feature for existing functionality. To complete the task multiple locations in the code base have to be located and identified. The task can be divided into three subtasks, that need to be accomplished in order to complete the whole task:

**Task 2.1** is adding a new menu item,

**Task 2.2** is adding a new Command-ID, and

**Task 2.3** is registering the new Command-ID with the CommandManager and providing the corresponding handler, that is called when the Command-ID is triggered by clicking on the menu item.

Each subtask consists of adding one line of code.
I limited the working time for this task to 25 minutes.

Participants have to highlight the lines of code and verbally outline their solutions.

In order to complete a task participants had to point out the lines in which a change or addition should be made and the participants had to verbally outline the changes or additions they would make in those lines. Only then was the task counted as completed. Moreover, the task was counted as successfully completed if the presented solution would result in a successful and working implementation and the correlating elucidations were compatible with the solution. The exact task descriptions can be found in appendix C.

A pilot test showed that code base and tasks are suitable.

A pilot test, that was conducted prior to the user study, showed that the code base was readable and understandable even if someone had very little or no experience with JavaScript, but at least basic knowledge about other programming languages. Moreover, the pilot test showed the appropriateness of the two tasks for this study: The term `jslint` has 651 occurrences within the code base and a search for the term `html` generates 4339 results. Hence,

[1]Issue 2950: https://github.com/adobe/brackets/pull/2950

the chance of finding the right solution for either task by chance and without understanding the source code is very small.

Sketchbar View was deactivated since there were not enough meaningful sketches and users of the pilot test took advantage of that.

In order to study the influence of an existing connection between source code and sketches on the participants' behavior, creating sketches and creating connections was omitted in the study. Hence, each participant of this study assumes the role of a new team member within a software project, who has to go through the onboarding process. As a consequence, the Sketchbar View was deactivated entirely for the course of this user study.

Another reason for removing the Sketchbar View from the user study was, that during the pilot test the Sketchbar View was mostly empty, due to the fact that the amount of available sketches was limited. Participants of the pilot test used the presence of sketches in a Sketchbar View as an indicator for being on the right track. One participant started to open nearly every Sketchbar View to check if the according file was needed to successfully complete the task. Filling some Sketchbar Views with meaningful, but task-unrelated sketches was not an option, due to the lack of available sketches and time constraints, that prevented the additional creation of meaningful sketches by a third and unbiased party. Therefore, during the final setting of the user study the focus was the Mission Control View.

A between groups study design was chosen.

A between groups study design with two conditions was chosen, so that every participant had to perform only one condition. Both groups were given the same

Both groups work with the same editor Adobe Brackets, ...

- **source code editor:** Participants worked with the default and, at that time, latest release of the open-source editor Adobe Brackets, i.e., Sprint 20. Adobe Brackets uses a third-party tool called JSLint[2] to ensure a certain quality within the code. JSLint is a static code analysis tool, that checks if the JavaScript source code complies with coding rules.

... the same code base, ...

- **code base:** The code base, on which the two tasks had to be performed, is the source code of Adobe Brackets

[2]https://github.com/douglascrockford/JSLint

Sprint 19. This version comprises about 214.000 lines of code in 1263 files distributed over 20 root folders. Except for the third-party and the extension folders, the code base has a very flat hierarchy with one or no sub-folders.

... and the same sketches that were provided by an unbiased third party.

- **sketches:** The sketches refer to parts of the code base and were created by an active Brackets developer, who works at Adobe. The sketches were created without any knowledge about the tasks. As a result the sketches contain drawings of things that are not related to the tasks in any way. In a sketch-enriched project a new team member would find many useful, but also task-unrelated sketches or even no sketches at all. Therefore, both the irrelevant and the relevant sketches were incorporated into the Mission Control View and the printed version, in order to provide a certain level of realism and to not be too obvious by design. The sketches and their arrangement can be found in the appendix D.

The difference between the two conditions is the way the sketches are presented as well as the availability of the software prototype, that provides the connection between sketches and source code in order to support the communication of source code design (see Figure 6.1):

The software prototype provides connections between sketches and code for the connection group.

**Condition 1** (connection group) provides access to the sketches only via the software prototype: All sketches were placed in the Mission Control View and connected to the source code. The connections were made and the sketches were places by a participant of the pilot test. A printed or any other version of the sketches was not provided to this group.

The control group has a printed version of the sketches.

**Condition 2** (control group) provides the sketches as a printed version on a DIN A3 sheet of paper with the same arrangement as in Condition 1, but lacking the link between sketches and source code. The Mission Control View was deactivated.

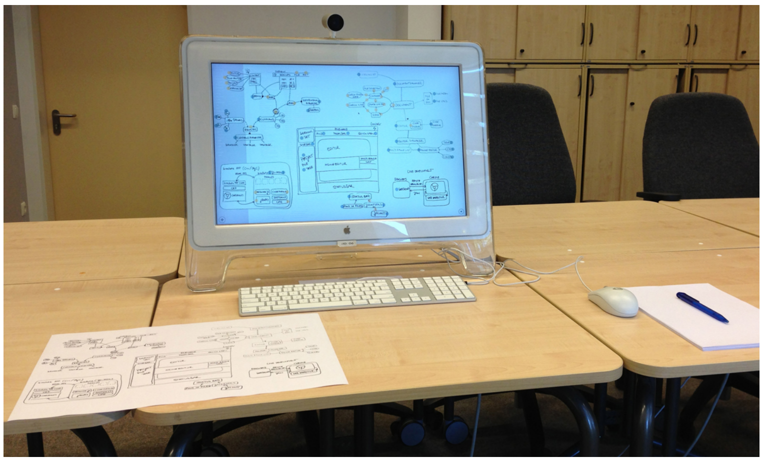


**Figure 6.1:** The setup used in the evaluation of the software prototype. Condition 1: The display shows the Mission Control View with the sketches. Condition 2: The lower left corner shows the same sketches and sketch-arrangement printed on a DIN A3 sheet of paper.

### 6.1.3 Semi-structured post-session interview

A post-session interview should provide subjective comments in order to verify H5.

In order to gather qualitative data with reference to H5, I prepared a semi-structured interview that deals with the hypothetical application of the software prototype in a project, that the participants are or were recently involved in. The second of two questions was formulated in an open way to support a natural and wide course of conversion and allow the participants to give comments freely without being lead by the question. The questions can be found in appendix B.

### 6.1.4 Methodology

Participants filled out a short pre-session questionnaire and were assigned to a condition randomly.

Participants were asked to fill out a consent form and a pre-session questionnaire, in order to assess their knowledge about JavaScript as well as Adobe Brackets and its source code (see appendix A). After completing the questionnaire each participant was assigned to one of the two conditions in the following manner: The first member of a pair of participants was assigned by chance and the second was auto-

However, participants with knowledge about the code base were distributed evenly among the two conditions.

matically assigned to the other group to ensure an even distribution of participants between the two conditions. The pairing of two participants, who would split up to either group, was conducted with regard to the pre-session questionnaire. The focus was to ensure, that the number of participants having knowledge about Adobe Brackets and its source code was evenly distributed among the two groups. Other than that no influence was exerted.

Participants were introduced to the editor, code base, and if necessary to the software prototype.

To begin with, each participant was given a short introduction to the code editor and its user interface. The default shortcuts for Find and Find Next were provided in written form. Moreover, the default third-party plug-in JSLint was described and explained, since it played an important role in both tasks. The software-prototype and the shortcut to toggle the Mission Control View was explained to all participants of the connection group. With regards to the sketches, participants of both groups were informed that the sketches had been created and arranged by a third-party.

The tasks were issued one after another and participants were asked to think aloud.

After answering general questions asked by some participants, the tasks were presented to the participants one after another, meaning that Task 2 was handed to the participants only after either the participant declared Task 1 as being completed regardless of whether it was completed successfully or not, or the maximum working time was reached, or the participant forwent the task. During the study, participants were asked to think aloud, in order to provide me with insights about their trains of thought and mental models while working. In order to promote thinking aloud, participants were allowed and encouraged to ask any questions, but questions were only answered if they did not reveal the solution for a task.

The Find In Project functionality was provided if it was asked for.

Participants were allowed to use the full range of functions provided by the default installation of Adobe Brackets. However, the Find In Project function was not mentioned during the introduction, but participants, who asked for such a function during the session, were provided with the corresponding shortcut.

All participants worked with the same hardware.

To ensure the same prerequisites, all participants worked with the same hardware, i.e., a MacBook Pro with a 2.4GHz

Intel Core 2 Duo processor and 8GB RAM. Participants used a 23″ screen with a resolution of 1920x1680 pixels, which is common for a modern work place for programming. Adobe Brackets was opened in the native Full Screen Mode of OS X version 10.8.3. Both the screen content and audio were recorded using Silverback[3] to allow further analysis afterwards if necessary.

After completing their tasks, participants were interviewed.

After participants completed the tasks, the semi-structured interviews were conducted. Participants of the control group were given a short demonstration of the prototype and were then asked the same two questions.

## 6.2 Results

### 6.2.1 Participants

32 participants including four professional software developers joined the user study.

A total of 32 participants joined the user study with an even split of 16 for each condition. 27 participants were male and five were female. Twelve participants were graduate and twelve were undergraduate computer science students. Another four graduate students were engineers or physicists with a background in programming and software development. The remaining four participants were professional software developers. The average age was 28 years with a maximum of 36 and minimum of 23 years.

21 participants had at least basic knowledge about JavaScript and four participants were familiar with the source code of Adobe Brackets.

### 6.2.2 Task Success

**Task 1:** This task was successfully completed by 14 participants of the connection group and ten participants of the control group. A comparison of the number of correct solutions for Task 1 showed no significant difference for the

[3]http://www.silverbackapp.com/

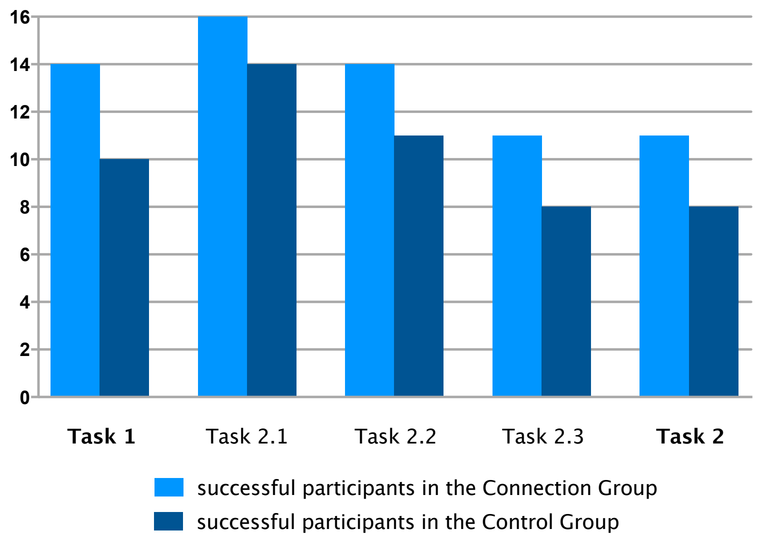


**Figure 6.2:** The figure shows how many participants were able to complete the tasks in each condition successfully. For each task and subtask, more participants of the connection group completed their tasks successfully.

two groups according to a two-tailed Fisher's exact test using the method of summing all p-values (p = 0.22).

**Task 2:** Regarding the three subtasks of Task 2, again, none showed a significant difference comparing the number of correct solutions according to two-tailed Fisher's exact tests using the method of summing all p-values (Task 2.1: p = 0.48; Task 2.2: p = 0.39; Task 2.3: p = 0.47). Taking a look at Task 2 as a whole, the task was considered to be solved successfully if all three subtasks were solved correctly. Therefore, eleven participants from the connection group and eight participants from the control group completed the second task successfully with no significant difference.

For neither task was the difference of correct solutions between the two conditions significant.

There were more participants in the connection group, who completed both tasks successfully then there were in the control group, but the task success rates were not significantly different. Consequently, H1 can not be confirmed.

### 6.2.3 Task Completion Times

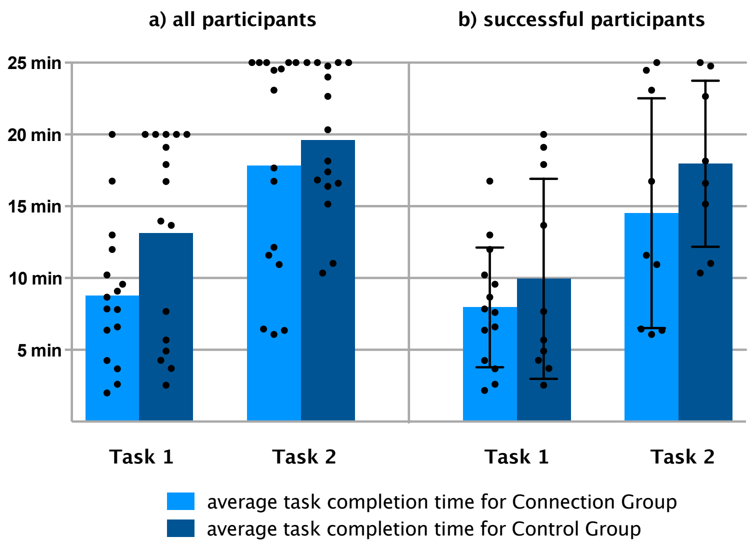


**Figure 6.3:** The figure shows the average time required to solve the two tasks: a) comparing measurements with and without the connection for all participants of the study, b) comparing measurements only for participants, who completed the task successfully.

**Task 1:** On average the successful participants of the connection group outperformed the successful control group participants by 1:59 minutes. But the difference between both groups was not significant according to an unpaired t-test with a Welch's correction (p = 0.43).

**Task 2:** The average participant of the connection group needed 14:31 minutes to successfully complete all three subtasks and thereby outperformed the average, successful control group participant by 3:27 minutes. Again the difference between the two groups was not significant (p = 0.32).

For neither task was the difference of the task completion times between both groups significant.

Participants, who could use the sketch-enriched code base and the software prototype completed their tasks faster, but not significantly faster. Consequently, H2 can not be confirmed.

### 6.2.4 Amount of Glances at Sketches

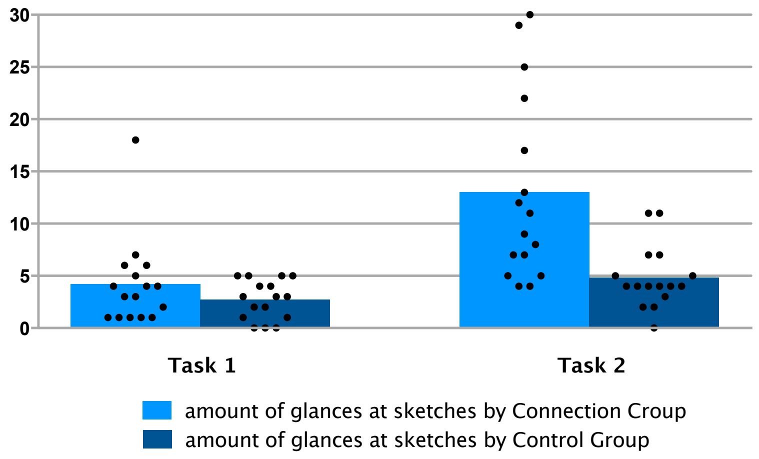


**Figure 6.4:** The figure shows the amount of glances taken at sketches by the participants while solving the tasks. The dots represent each participant. The bars represent the average amount of glances.

**Task 1:** On average the participants of the connection group looked 4.2 times at the Mission Control View with a total of 67 times. Whereas the participants of the control group looked at the printed version of the sketch 2.7 times with a total of 43 glances. Despite the 56% increase of glances there was no significant difference according to a Mann-Whitney test (p = 0.33).

**Task 2:** During the second task participants of the connection group looked at the Mission Control View 208 times in total with an average of 13 glances per participant. In contrast the participants of the control group looked at the printed version of the sketch 77 times with an average of 4.8 glances per participant. Since the values of Task 2 for each group passed the D'Agostino & Pearson omnibus normality test (connection group: K2 = 2.7, p = 0.25; control group: K2 = 3.9, p = 0.14), the unpaired t-test with Welch's correction was applied to the data and shows a significant difference (t = 3.483, df = 18.29, p = 0.0026).

Participants of the connection group looked more often at the Mission Control View than participants of the control group looked at the printed version of the sketch. The difference was significant for Task 2, but not for Task 1. Consequently, H3 can not be confirmed in general.

The difference of glances taken by each group is significant for Task 2, but not for Task 1.

### 6.2.5 Time Spent Looking at Sketches

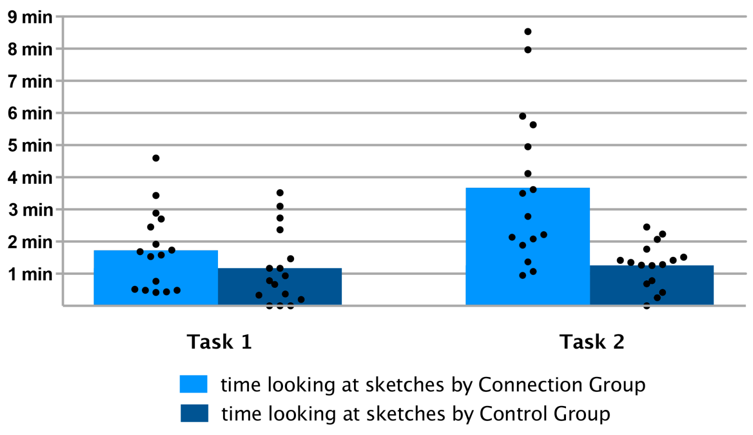


**Figure 6.5:** The figure shows the duration of glances taken at sketches by participants while solving the tasks. The dots represent each participant. The bars represent the average duration of glances.

**Task 1:** Participants in the connection group looked at the Mission Control View for 1:44 minutes on average. Compared to the control group, where a participant looked for 1:11 minutes on the sketches, no significant difference was detected (unpaired t-test with Welch's correction: p = 0.20).

**Task 2:** During this task the participants in the connection group looked at the Mission Control View for 3:40 minutes on average, i.e., 20.3% of the task completion time of all participants, whereas the participants of the control group looked at the sketch for 1:15 minutes on average, i.e., 5.9% of the task completion time of all participants. Since the values of this task passed the D'Agostino & Pearson omnibus normality test (connection group: K2 = 2.41, p = 0.92; control group: K2 = 0.16, p = 0.30) for each group, the unpaired t-test with Welch's correction was applied and showed a significant difference (t = 3.923, df = 17.60, p = 0.0010).

The difference of the overall duration of glances taken by participants of each group is only significant for Task 2.

Participants of the connection group looked at the Mission Control View longer than participants of the control group looked at the printed version of the sketch. The difference was significant for Task 2, but not for Task 1. Consequently, H4 can not be confirmed in general.

### 6.2.6 Qualitative Observations

Distinct differences in the behavior of the two groups were observed.

For either group very clear patterns of behavior were observed during the sessions. The following two segments will provide a brief insight into how the average participant made use of the sketches resp. the sketches and their connections to the sketches depending on the group affiliation, based on my observations during the studies.

#### Control group Members

Control group members used the sketch rarely and relied on well-established methods like scrolling, clicking and searching to complete the tasks.

The participants in this group read the task description and then looked at the sketches provided on a DIN A3 piece of paper. They studied each and every sketch on the paper to find potential hints on where to start the task. Since the participants were asked to think aloud, most participants stated that they could not find anything helpful, so they started to work with the editor. Eleven participants moved the paper with the sketches farther to their left and put the paper with the task description right in front of them, so that the sketches disappeared from their field of view. Working with the editor, participants used an already well-documented set of operations (Ko et al. [2006], Starke et al. [2009]). The main operations that were performed were:

- **navigating via project tree** to find, re-find and open files
- **navigating via tabs** to quickly switch between already opened files
- **scrolling through files** to find useful information by skimming the code
- **reading comments** of files and methods
- **searching** within a file or within the whole project

Participants glanced at the sketch when reaching an impasse, but quickly turned back the the editor.

If participants reached an impasse, they would take another look at the sketches to check, for they might have missed some important clues or information that could have helped them to complete the task. At the very end of a task some participants would take another look at the sketch to check if they might have overseen something, e.g., participants pointed out the correct line in the code and described the correct changes they would apply verbally and took a quick look at the sketch to see if they might have forgotten something.

During the second task the behavior was quite similar, however, after the initial skimming of the sketch and searching for parts regarding this task, participants tried to reassure themselves of their correct approach during the task about two to four times by looking at the sketch. These glances were initiated by statements like

> "I will take another look at the sketches, since they have been provided, ... there should be something on them."

> "Oh, I completely forgot the sketches, maybe they will help me now ... No, still not helpful."

> "Let's take another look ... I got that and I got that, but I don't know where to find that

> ... I think I found something on the sketch, that might be helpful, so I will try to find that in the code by searching for that exact term provided in the sketch."

> "Let's have another look ... OK, I see that this sketch has to do with my task, but I don't get it, so I'll resume working with the editor."

One participants did not use the sketches at all.

One participant stood out by not looking at the sketches at all. This participant had basic knowledge of JavaScript and no knowledge about the code base. The task completion time for Task 1 was 7:40 minutes and 10:21 minutes for Task 2. Other than that, the behavior did not deviate from other participants', who looked at the sketches. After asking him for the reason for not looking at the sketches he replied that he had not noticed at all that he did not look at the sketches.

Sketches were mainly used as a last resort to find information.

All in all, the sketches were used as a reference and as a tool to reassure the participants actions, but mainly as a last resort. Due to the fact that most participants looked at the sketches since they were provided as part of the task and not because they felt the need to, participants used the aforementioned common methods to solve the tasks and understand the code.

### Connection group Members

Connection group members used the Mission Control View and the connections to source code quite often and started to toggle the view frequently.

Participants of this group read the task description and then opened the initially closed Mission Control View. Similarly to the control group, participants in the connection group took the initial glance to get an overview of all sketches. Task 1 clearly was the task where participants accustomed themselves with the Mission Control View and the functionality provided. While it was not very obvious during Task 1, the behavior of this group clearly shifted during the second task: Participants constantly switched between the Mission Control View and the source code in order to navigate the code base. It quickly became clear, that folder and file names were not that important, since elements of the Mission Control View turned out to be suitable substitutions even if the names of the elements did not coincide with the filename or the method names they were connected to.

Participants of the connection group mixed the use of the Mission Control View with other well-established methods constantly.

The aforementioned operations that were used by the control group members, however, were not substituted, but rather complemented by the navigational properties of the Mission Control View. However, as soon as the participants felt, that the sketches and the connections provided by the Mission Control View, would not help them, they fell back into old habits for a short amount of time and, e.g., started to search within files as well as the whole project and navigated via the project tree or tabs, only to come back to the Mission Control View and use its functionality again to continue with the task.

The combination of sketches, comments, and meaningful method names was helpful.

With the help of the Mission Control View, participants were able to find the correct lines within the code quite fast, but did not realize their success at first. Together with method names and comments of methods and files, I was able to observe the formation of the mental model.

A few participants stated that they would not have created some of the connections provided, but rather connected different lines of code or files with the sketches. Interestingly, some of those participants withdrew their statement during the task by saying

> "Now that I understand the concept, I guess it makes sense to connect these particular lines of code with that sketch. I'm not sure if it is the best way to do it, but it's OK" or words to that effect.

Participants used the Mission Control View to recover their train of thoughts.

Another common observation was that participants were scrolling through a file and reading comments, when suddenly they asked:

> "What was I looking for again?"

and immediately opened the Mission Control View to find the highlighted File Dot in order to see, where they were with regard to the sketches, whereas in the same situation most participants of the control group turned to the task description and not to the sketches provided on paper.

Most participants explained the mental model behind the source code to themselves several times.

It is particularly noteworthy that most participants either partially or entirely explained the way they understood the individual subtasks and how they worked together to themselves during the session in order to recapitulate their progress in some way. Before they gave their final answer in order to successfully complete the task, they mentally walked through their individual steps with words like

> ”So I added the menu item here in line 128 and provided the Command-ID that I decelerated in the Command.js. Now I want to register that Command-ID with the CommandManager and I obviously have to use the register method for that. I have all parameters, but the function that is executed when I click on the menu item and I don’t know where this call of the register method has to go” or words to that effect.

This recapitulation of the progress was made with the Mission Control View being opened and by pointing on the sketches and following the sketched lines as well as clicking onto the Connection Dots to get to the corresponding code segments to prove to themselves, that they had considered every part of the task. This kind of behavior was not observed with the control group.

Two participants used the Mission Control View very extensively and provided a lot of useful insights.

One professional software developer and one graduate industrial engineer working at an IT company embraced the functionality of the Mission Control View by extensively using the connections from the sketches to the source code. They had the highest amount of glances at the Mission Control View (29 times and 30 times). With an average glance duration of 16 seconds, one third of their task completion time consisted of using the Mission Control View as a way to navigate through the code base. Whereas some participants showed only few signs of a change in their way to approach the tasks, the change of behavior for these two participants was very evident compared to the behavior of control group members.

In conclusion, the Mission Control View and the connections between sketches and source code were used in the following scenarios:

Four main behavioral scenarios emerged: navigation via Connection Dots, review of context, orientation within the context, and mental walkthroughs.

- **navigation:** In order to jump to a certain part of the code base the Mission Control View was opened and a Connection Dot was used. This way of navigating was liked by all connection group participants and reported as very helpful by eleven participants of the connection group.
- **context comprehension:** The Mission Control View was opened and looked at in order to review which other parts were related to the current element of interest. This was reported as something unprecedented in this form (five participants) and extremely helpful (twelve participants).
- **orientation within the context:** Highlighted Connection Dots were used as an indicator for the current position within the context. All participants used this feature by very quickly toggling the Mission Control View.
- **mental walkthroughs:** The Mission Control View was used as a way to be reassured by self about the validity of the approach chosen to complete the tasks and to check if something was omitted. This was observed with every participants of the connection group, but in many different ways: Some participants argued with themselves, whereas others mumbled to themselves or just pointed on the Mission Control View with their finger and navigated through the sketch while going through the steps mentally.

Another observation was that only two participants used a back-link from the source code to the sketches in the Mission Control View by clicking on the marked line numbers in order to see to what sketch that code segment was connected. Each of those two participants used the back-links

Back-links were not used, supposably because there were to few in the code base and the Mission Control View was sufficing.

once. This extremely small number might be explained by the fact that on the one hand the approach to the Mission Control View was a new concept and on the other hand the relatively small number of back-links within the code base may not have encouraged the use of said back-links. Another possible explanation are the Connection Dots that were connected to files: While being in a file that had a connection to a sketch the corresponding Connection Dot was automatically highlighted in the Mission Control View. This seems to have been a sufficient alternative to back-links, hence, participants opened the Mission Control View and could immediately orient themselves without using a back-link. But, with an activated Sketchbar View and more back-links to sketches the usage of back-links might have been higher and revealed additional insights.

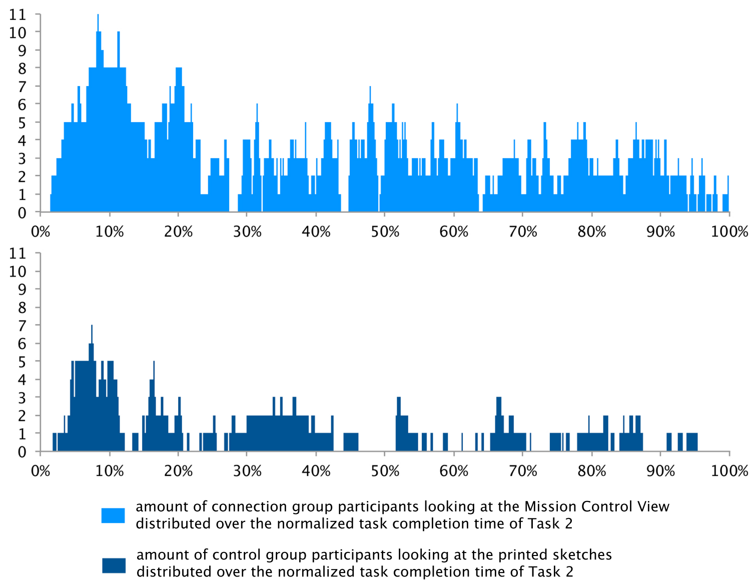


**Figure 6.6:** The figure shows the amount of participants looking at the sketches distributed over the normalized task completion time of Task 2. The initial skimming of the sketches observed in both groups is represented by the peaks at around 10%. The bar chart for the connection group shows a more frequent usage of the sketches due to the source code connections provided by the Mission Control View conveying the observation that the Mission Control View was used for navigational purposes.

In order to visualize the difference of behavior and interaction with the sketch, the task completion times were normalized for each of the 32 participants. Figure 6.6 shows that the way and frequency in which the sketches were used during the second task was quite similar for all participants of either group. While most participants of both groups looked at the sketches after reading the task description (peaks at around 10%), participants of the connection group continued the use the Mission Control View as well as the sketches and the connections to the source code during the second task. Consistent with observations made during the user study, the bar graph depicts that the sketches provided via the Mission Control View were consulted more frequently and by more participants compared to the control group with its printed version of the sketches.

Participants of the connection group used to look at the Mission Control View more frequently and evenly distributed.

### 6.2.7 Semi-structured Post-Session Interview and Participants' Comments

After the test, I initiated the interview session with two questions (see appendix B) so that an informal chat could be initiated. 15 participants were working on a solo project and another 15 participants were working on team projects at the time of the interview. Two participants were working both on a team project and a solo project. Moreover, participants were asked if and how they would imagine to use the functionality of being able to connect sketches with source code with regard to their projects.

15 participants worked in a solo project, 15 in team projects, and two in both.

#### Areas of Application

Regardless of their group affiliation, participants imagined the prototype to be useful within their project in the following areas:

Five areas of application emerged during the interviews.

- **navigational support:** Participants of the connection group liked the idea of navigating through their project via the Mission Control View using the Connection Dots. Some instantly imagined their project affiliated sketches and visualizations and were exited to connect them to the source code asking how sketches can be imported into the Mission Control View and how connections to the source code are created. Participants of the control group imagined that the Connection Dots can be helpful since most participants reported that the printed version of the sketches was not very helpful in finding the correct files or code lines and they had to use the search function instead.

- **project overview / software architecture:** Participants of both groups imagined the Mission Control View to provide an adequate overview of the project and the software architecture. Participants also reported that they would use the Mission Control View to conceive the context of the task they were working on, to see which other team members had to be involved in the task or which other parts of the project had to be considered. Participants of the connection group reported that the Mission Control View, was a way to not loose track of the task at hand by "zooming out into a kind of meta view".
- **documentation / design decisions:** Sketches and visualizations like UI elements or informal class diagrams of the software architecture created in different project phases could be collected and stored in one place as part of the documentation.

  > "Sketches and the connection to code are not enough documentation in my opinion, but together with comments it would be pretty awesome. If I wanted to know more about a certain part of the project that was somehow connected or close to my part of the project, then I could just click on it and then read some comments to get an overall understanding."

  > "When I return to a certain part of the code I worked on like two weeks ago, I can't remember why I did certain stuff and how the classes and methods work together. I knew it two weeks ago like the back of my hand,

> but now ... nothing! I normally take a look at my sketches, that I made at that time or if I already threw them away I start to assemble all the missing parts in my head by walking through the source code and sometimes making new drawings. But, this is really tedious and sometimes even the reason why I don't make changes to the code anyway, although I know it would be better to refactor and change my code. So in such a case, the Mission Control View and the links to the code would be very helpful" or words to that affect.

- **capturing the development progress:** Some participants had the idea that the Mission Control View could be used as a Manager's View meaning that a project leader could see the progress of the project, i.e., new elements that had been added to the view or changes that had been made. The project leader could jump into the corresponding part of the code and have an insight into the work that is already done. Two participants imagined adding 'changed since last visit'-indicators and an overview with a timeline, so that it is possible to scroll through the progress of the project and see the development of new elements and the change of existing elements and their relationships amongst each other:

  > "I imagine it like Apple's TimeMachine[4], but in fact it would be the data from the version control system. Because, let's be honest: Who likes to read commit messages? That would really be a fun way to oversee the project's development."

- **onboarding process:** Above all, participants imagined this functionality to be very helpful for new team members. They reported that it is hard for a new team member to catch up with all the knowledge about the project and the decisions that have been made during the design process and the implementation phase. Such a view on the project would be an enormous

[4]http://www.apple.com/osx/apps/#time-machine

support to get to know the project. One professional software developer mentioned that they had special projects made for new team members, that are meant to help the new team members to familiarize themselves with the project without being at risk of generating any damage to the productive version of the project:

> "Oh, I can see that implemented in our sample projects and be helpful to new coworkers. Since we already spend time creating these sample projects, adding the connections between sketches and the source code manually wouldn't be that tragic... as long as the cost-benefit ratio is right, I guess."

Twelve participants regularly create sketches, three only use visualizations, nine create both and four create neither.

During conversations the topic of sketches and visualizations as well as their creation came up. Twelve participants reported that they create sketches while working on their project, three stated to solely use visualizations created with tools like Omnigraffle[5] or StarUML[6], and nine participants reported to sketch as well as use tool-based visualizations depending on the situation or the project phase they were in. Secluding, four participants reported to neither sketch nor use tool-based visualizations at all. Reasons for participants not sketching in their project were, that the project was to small to sketch or they could maintain all the mental work in their head and had no need for externalizing their thoughts.

Participants archive sketches and visualizations if they anticipate potential future use.

Moreover, participants stated that they mainly sketch during initial phases of their projects. Some participants confirmed that they keep their drawings or digitalize them, e.g., by taking a picture with their smartphone if they saw potential future reusability. If the sketch was important enough participants would create a tool-based visualization to have a clean version of the sketch. Participants using only tool-based visualization reported that they did not like to sketch, since it was too messy and their drawing skills were not refined enough so that their sketches were rather

[5] http://www.omnigroup.com/products/omnigraffle
[6] http://staruml.sourceforge.net

hard to decipher for others or self after a certain amount of time or the thoughts they had during the creation of said sketches were no longer of concern. These findings coincide with the findings reported of Branham et al. [2010], Cherubini et al. [2007b], and Walny et al. [2011].

### Identified Problems

Four major problems emerged while talking about implementing the prototype within the participants project.

However, participants of the groups also identified problems and challenges with the application of the functionality of connecting source code and sketches within their projects:

- **creation of sketches / visualizations:** The creation of sketches was identified as a problem by almost every participant. Despite the fact that most participants created sketches or visualizations, they still mentioned that creating sketches is time consuming. Sketches created during team meetings or adhoc meetings of two or three team members are valuable byproducts, but the creation of a sketch for the sketches sake was recognized as an additional burden. It was compared to documenting the source code by some participants:

  > "I guess creating sketches for a project is a nice and helpful thing, but it's like with documentation: You know you're supposed to do it, but you still don't do it".

- **standardized sketching:** Some participants considered the quality of their own sketches and were concerned about the readability of sketches. They imagined that every team member of their project would contribute and provide sketches and they saw potential problems in how helpful these sketches were if they came below an acceptable level of quality.

  > "What's the use anyhow if I am not able to recognize parts of a sketch because they are too scribbled."

> “What if I used rectangular elements for some thing and a teammate used circles for the same thing?”

With this in mind, some participants even mentioned that they often were not able to decipher their own sketches after a certain time. This problem would definitely lower the value of the whole functionality since the sketches are a very important part of the connection. As a possible solution, participants suggested formal conventions for sketches that applied to the whole team.

- **currentness:** Since very few participants reported to re-sketch their own sketches if they were not up-to-date, the foremost mentioned problem was the currentness of sketches and connections.

  > “What happens if I change something in the code? Oh ... Do I really have to re-sketch this part? Then I would probably leave the sketch as is.”

  Participants were torn between the fact that they would like to have the connections as well as the sketches created automatically and the fact that sketches had a “certain charm” of their own and were a ″prove of mental work”, as one participant phrased it. Participants, who created solely tool-based visualizations, reported that they would simply change their visualizations with the corresponding tool and then add the new visualization to the Mission Control View by replacing the old one. Participants stated that they would predict the enthusiasm to maintain sketches and connections to flatten with time and that the “new feature”-status of the functionality would fade.

- **team-size:** All participants agreed that keeping the sketches and connections alive might be realistic for a rather small team of developers and a medium-sized project. Larger teams would have problems to maintain the sketches and a certain level of quality. Participants, who worked in solo projects, liked the functionality provided by the prototype, but agreed that creating and maintaining the sketches as well as the connections in addition to creating and maintaining the source code, which is the main task of software development projects, was too much of a burden for one person and seen as overkill.

Participants had to wait after clicking on a Connection Dot too long.

Another concern some participants had was about performance: The delay between clicking on a Connection Dot and finally seeing the connected file or line of code was reported as too long by some participants. Although participants understood that a research prototype may suffer from problems like this, they still found themselves hindered in navigating more quickly sometimes and started to double and triple click a Connection Dot if the performance got bad.

One participant was currently in the onboarding process at work and provided first hand knowledge.

One participant in particular had a special view at the functionality and the software prototype. She recently joined a software development team and was currently in the onboarding process. She told of interrupting her mentor constantly and taking notes during these short ad-hoc meetings, but she was not used to sketching and therefore drawing sketches was not one of her strong points. During the post-session conversation she said:

> "I see how sketches and the connection can be helpful. The sketches were like a road map to me. I think using such a map is easier than searching because you don't need to know exactly what you are looking for. The sketches can complete the missing parts or even tell you what to look for. I think I will start sketching more and archive those sketches. Maybe I can create such a map for our project at work and it could make things easier for the next new team member... Is there a way to connect sketches to source code in the IDE we use at work?"

### Suggested Ideas

Ideas for enhancing the software prototype: tabs and intelligent zoomable interface

Three participants had own ideas on how the software prototype might be enhanced and further developed. One idea was to add tabs to the Mission Control View. Each tab could hold the same sketches but users would be able to rearrange them as needed, e.g., for different tasks or different users. Moreover, these tabs could act as views at the system with different levels of detail. Another user imagined an area zoomable pane, meaning that within one view there could be multiple zoom levels in different areas of the view, e.g., one area depicting component A could be zoomed to a more detailed view and another area depicting component B could be zoomed out completely in order to show the basic structure. Being able to add connections between two or more sketches would enhance the possibilities of conveying the concept and structure behind the source code. Connection Dots would be merged and information would be aggregated in a zoomed out view whereas the Connection Dots would split and show on the corresponding sketches with a higher zoom level. Some ideas mentioned by these participants are similar to approaches taken in the field of zoomable user interfaces and off-screen visualizations (Bederson and Hollan [1994], Bederson et al. [2000], Baudisch and Rosenholtz [2003], Zellweger et al. [2003]).

### Implications for H5

The software prototype might be the reason for the connection group outperforming the control group even though they were looking at sketches more often and longer.

Although, both the task success ratios and task completion times were not significantly different when comparing the two groups, the fact that the connection group participants looked at the sketches for 20.3% on average during Task 2 and still outperformed the control group in both tasks, suggests that the Mission Control View and the connection between sketches and source code may have contributed to that affect.

In conclusion, there was a consensus among the participants of the connection group that the connection between sketches and source code had helped them understanding source code designs. This was noted especially for Task 2 and could be observed during the mental walkthroughs. Participants of the control group imagined that the ability of jumping through the code via the Mission Control View would be helpful, but some control group participants were skeptical if the navigational element of the Mission Control View would have helped them understanding the mental model, since they gained that understanding although not using the software prototype. Consequently, H5 can be confirmed partially, but not in general.

Comments made by connection group participants suggest that the connection between source code and sketch is helpful for understanding source code.

## 6.3 Discussion

Taking a look at the tasks presented in the user study and the quantitative data gathered in order to support H1-H4, the difference between the two tasks is clearly visible: Both groups used the first task to acclimatize to the environment of the user study, i.e., the source code editor, the code base and the sketches. However, during the second task participants were more engaged and felt more comfortable. Especially connection group participants used the Mission Control with ease, after getting used to the functionality during Task 1.

Task 1 was used to get acclimatized.

In terms of appropriateness, the first task had a very narrow solution path and did not provide enough opportunity to use the knowledge gathered through techniques like searching, skimming, and scrolling the source code and for the connection group additionally the use of the Mission Control View. My observations of the participants showed that the knowledge gained during the first task was applied mainly during Task 2.

Task 1 was a prelude for Task 2.

Therefore, it is my opinion that the tasks were appropriate in combination and Task 1 was a prelude to Task 2, especially for the connection group, since they had to become acquainted with the Mission Control View. Task 1 was dealing more with syntactical knowledge and Task 2 with semantical knowledge. This may have contributed to the results and insignificant differences for Task 1 regarding H3 and H4.

The tasks were appropriate in combination.

Amount of glances of the control group were low since the sketches were put out of sight.

With regard to the usage of sketches, I affiliate the observation that participants of the control group did not use the sketches as often as the connection group to the fact that the printed version of the sketches was often put aside and, therefore, not in the participants' line of sight. Before each session the printed version of the sketches was placed directly in front of the participant, i.e., between the participant and the keyboard (unlike seen in Figure 6.1 the sketch was placed more to the right). Participants put the sketches aside upon receiving the description for Task 1 and in most cases the sketches stayed to the participants' left side even if they searched for more information to complete a task.

Connection group could access the sketches easily through the Mission Control View, without changing the field of vision.

In contrast, participants of the connection group looked at the Mission Control View when searching for more information. I affiliate this observation to the ease of use of the software prototype and the ability to quickly switch between the content area containing the source code and the Mission Control View, without having to change the field of vision and move the head. I conclude that the Mission Control View can be interpreted as an integrated and utilizable source of information, whereas the printed version can be seen as an additional, but not directly utilizable source of information.

Mental walkthroughs suggest build up of mental model.

In conclusion, for a first software prototype, participants found the functionality to be very helpful while navigating the code base. Usability issues were reported and valuable feedback for future designs was obtained. The mental walkthroughs observed in the connection group suggests that participants were building up a mental model of the source code design regarding the implemented concept of menus in the source code of Adobe Brackets (Task 2). Conversations with the participants of both groups suggest that the functionality to connect sketches with source code in order to communicate source code designs can be especially helpful during the onboarding process.

# Chapter 7

# Summary and Future Work

This thesis addresses the communication of source code designs through sketching with the goal to support software developers and programmers understanding the mental model of source code. The approach presented in this thesis is to connect source code, which is a very low-level source of information, to sketches that can be anything from low-level details to high-level concepts about the source code. Thus, enabling programmers and software developers to use sketches in order to navigate and understand the source code.

## 7.1 Summary and Contributions

Software comprehension models describe how a mental model is formed.

To provide an insight of how programmers try to understand source code and its behavior, I started by outlining software comprehension models and the fundamental elements involved. The basic idea behind software comprehension can be compared to a black box called the assimilation process that has external representations of the source code, a programmer's knowledge base, and the programmer's current mental model of the source code as an input and the output is a refined and updated mental model of the source code.

Sketches can depict different levels of abstraction and can support many different strategies for the assimilation process.

Software comprehension strategies suggest that both low-level information as well as high-level abstractions are of value during the process of understanding. However, the information and the level of abstraction that is helpful is different in each situation and for each person. That is why software developers often create sketches while explaining the source code design to self or others. Sketches have the ability to provide the level of abstraction that is actually needed at that time and, moreover, can provide different levels of abstraction at the same time. Whereas computer-based tools can only visualize information that is existing in the code, sketching offers the ability to sketch exactly what needs to be visualized. Sketches are a very versatile tool that can support many software comprehension strategies, since a sketch can depict anything that is needed to support the assimilation process, regardless of whether, e.g, the data flow or the control flow is used to create a mental model.

Sketches are an ideation tool, support communication with self or others, and can depict more than what source code visualizations provide.

Reviewing related work in the context of this thesis showed that sketching supports the ideation process through ambiguity and incompleteness of sketches. Moreover, sketches are a very helpful means of communication, since they provide an additional way to better understand discussed issues and fill the gaps that spoken language may create. In the context of software development, programmers reported that sketches are important when trying to understand existing source code, designing/refactoring, and during ad-hoc meetings. Source code visualizations were not important in these three scenarios, even though these external representations are automatically generated without any assistance of the programmer. Research also showed that sketches are archived and reused by programmers and some tools have been introduced on that account.

An initial study provided first-hand experience about sketches in the software development process.

To deepen my understanding of how software developers and software architects use sketches and drawings in their everyday work, I visited an IT-company that focusses on solutions for the energy industry and observed two software architectural meetings and interviewed members of the software development team. The results from that first-hand experience are that the team members' knowledge is mainly in their heads, sketching is used as a tool of communication to externalize thoughts and ideas, and that sketches can be very helpful during the onboarding process of new team members.

Fundamentals about the connection between source code and sketches were introduced.

With the literature review and the initial study in mind, the fundamentals of a connection between source code and hand-drawn sketches were established in chapter 4. I presented basic layout techniques within IDEs as well as views for sketches within these layouts. Moreover, I explored the Connection Points of both source code and sketches which are the termini of a connection and I presented two categories of connections between Connection Points, i.e., visualized connections and indicated connections.

The prototyping process resulted in a software prototype incorporating two different layouts.

In order to be able to evaluate the approach, I started exploring the connection between source code and sketches on paper and implemented a software prototype, hereafter. During the prototyping process constant feedback and analysis sessions focussed and streamlined the prototype with each iteration. The final prototype was implemented as an extension for the open-source code editor Adobe Brackets and offers two different views: The Mission Control View is a fullscreen overlay that provides an map-like overview of the underlying source code. The Sketchbar View is a canvas in a side panel next to the source code and also implements a Map View. In particular, the challenges concerning the navigational behavior of the Sketchbar View, were explored and situation-dependent synchronous scrolling was implemented to support an intuitive interaction. However, this navigational behavior of the Sketchbar View was only possible due to the single, continuous selection metaphor of the used source code editor.

I evaluated the software prototype and studied the connection between source code and sketches.

To evaluate the functionality of connecting hand-drawn sketches and source code, I conducted a between groups user study and for the duration of the user study only the Mission Control View was available to the participants. I wanted to know, if the connection of hand-drawn sketches and source code would communicate source code designs to the participants and, therefore, support their software comprehension process. The results of the evaluation showed no significant difference in the task success rates or the task completion times, despite the fact that participants who used the software prototype outperformed the participants of the control group in both cases. However, participants who used the software prototype looked significantly longer and significantly more frequent at sketches on average and still outperformed the control group. Observations and comments suggest that the connection between source code and sketches is helpful. The prototype was used to navigate the code base, comprehend the context, orientate within the context and to support mental walkthroughs. Mentioned benefits of the functionality were the ability to connect any visualization (sketches and tool-based visualizations) to source code without restrictions as well as the ability to integrate new sketched ideas and features into the context of the source code and start from there. Problems that were identified were the burden of creating sketches manually as well as keeping the sketches up-to-date and readable despite the lack of sketching conventions.

The contribution of this thesis.

To sum up, this work contributes the investigation and exploration of the connection between source code and sketches. Furthermore, this work gives empirical indications for supporting the claim that the connection of sketches can convey source code design better (not necessarily faster) than sketches without a connection to source code. I conclude that an approach integrating sketches into the IDE and connecting them to source code can be especially helpful during the onboarding process of new team members joining a software development project.

## 7.2 Future Work

The functionality of providing a connection between sketches and source code implemented in the software prototype was well received by the participants of the user study. Future work in this direction should consider the benefits, but also the shortcomings presented in this thesis.

Provide more automatization and assist the user.

An enhanced prototype should provide more automatization and assistance: Creating and maintaining connections manually was identified as a burden and clear disadvantage. But, in order to still be able to use the benefits of sketches and sketching the integration of hand-drawn sketches should not be omitted completely. I suggest a

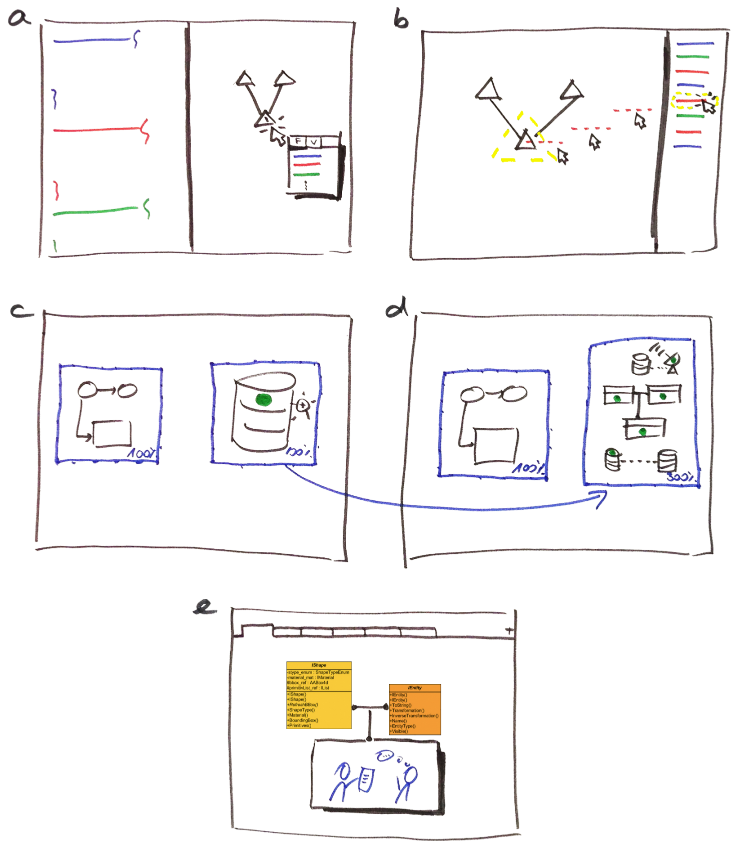


**Figure 7.1:** These sketches depict suggestions for future software prototypes: a) select semantical elements via a context menu to connect them to sketches, b) drag and drop semantical elements from a list to connect them to sketches, c+d) "zooming" between different layers of the same concept to provide high-level concepts and low-level details, and e) tabbed Map View and source code visualizations combined with hand-drawn sketches to support better maintenance.

hybrid approach that combines re-engineering capabilities and code independent visualizations. To facilitate the burden maintenance of connections the Connection Points of source code should be semantical elements, e.g., variables and functions, rather then syntactical elements, e.g., lines and blocks (see Figure 7.1a+b). In addition, sketches could share the space with source code visualizations, in order to provide a more holistic experience and bring these two ap-

Mix sketches and source code visualizations.

proaches closer together (see Figure 7.1e).

Tabs and different layers of abstraction.

Since the Map View used in the software prototype was reported to be useful, future work should allow for map-like arrangements to support project orientation and navigation. To enhance such a view, participants suggested additional features like tabbed Map Views for different work situations and tasks of one person, but also to provide views for multiple users involved in a project. Moreover, "zooming" within different layers of the same sketch, but with different levels of abstraction could enhance fluid transitions between high-level concepts and low-level details (see Figure 7.1c+d).

A zoomable Map View in a Split Layout Layout.

With regard to a Split Layout in which source code and sketches are displayed side-by-side, future work should evaluate the capabilities of the view itself, regardless of whether a Map View or a Context View is chosen. Adding a zoomable user interface to a Split Layout might add navigational advantages, and, therefore, should be explored (see Figure 7.2).

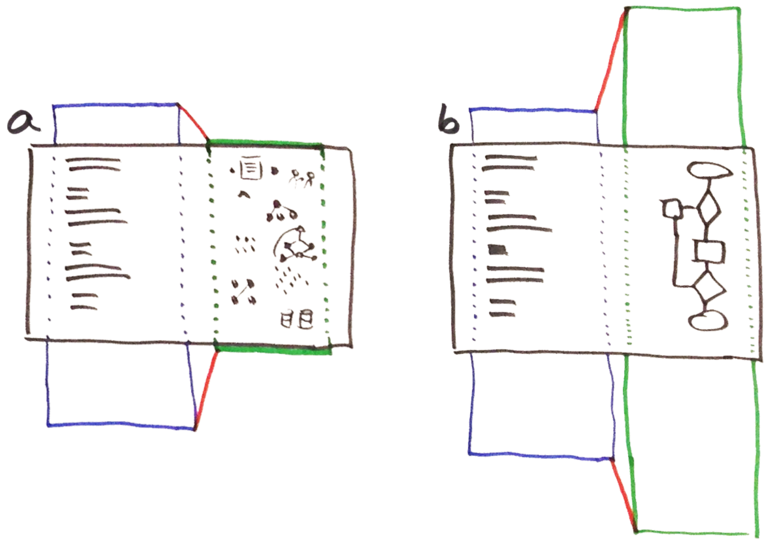


**Figure 7.2:** A suggestion for a future Split Layout approach: The Map View zoomable and scrolling is relative and dependent on the zoom level.

# Appendix A

# User Study: Declaration Of Consent & Pre-Session Questionnaire

ID

C

## Informed Consent Form

Evaluation of connection between source code and sketches
Principal Investigator: Lukas Spychalski, Media Computing Group, RWTH Aachen University

**Purpose of the study**
This study is conducted in relation to my diploma thesis at RWTH Aachen University.
The goal is to study if the connection between source code and sketches can be an additional channel of communication in order to enhance the understanding of source code.

**Procedure**
You will be asked to perform two tasks in which you should navigate through source code. You will be asked to answer some questions. The study will take up to 40 minutes.

**Risks / Alternatives to Participation**
There are no risks associated with participation in the study. Participation in this study is voluntary. You are free to withdraw or discontinue the participation at any time.

**Confidentiality**
All information collected during the study period will be kept strictly confidential. You will be identified through identification numbers. No publications or reports from this project will include identifying information on any participant.

☐ I have read and understood the information on this form

☐ I agree to being filmed during the study (screencapturing & face via webcam)

________________________________________
date / participant's signature

---

| | |
|---|---|
| gender | ☐ male ☐ female |
| age | ________ years |
| occupation | ________________________ |
| programming experience | ________ years |

Do you know about JavaScript? ☐ yes ☐ no
Do you know the source code of Adobe Brackets? ☐ yes ☐ no

---

☐ I want to enter the lottery for a 50€ Amazon gift card

____________________________________
email address

# Appendix B

# User Study: Post-Session Interview Questions

Please think of a project, that you are currently working on or recently worked on.

Did you work on that project by yourself or in a team? ☐ solo ☐ team

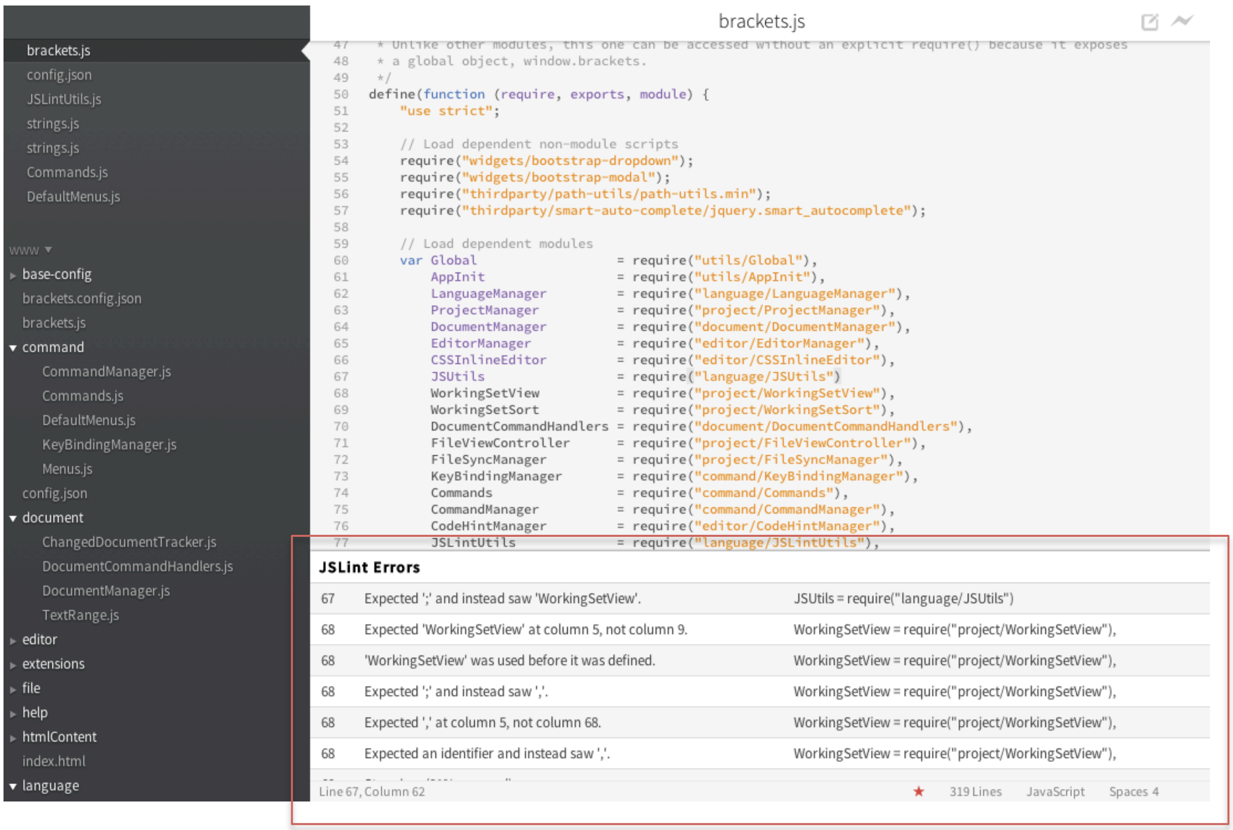
How and what for would you use the tool in that project?

# Appendix C

# User Study: Task Descriptions

Brackets uses JSLint (JavaScript Code Quality Tool) to maintain a certain quality in the code. JSLint is executed automatically when the file is saved and the occurring errors are listed in the statusbar. In the given implementation of Brackets both JavaScript files (.js) and HTML files (.html/.htm) are checked by JSLint. Checking HTML files with a JavaScript Quality Tool is a bug.

Example of how JSLint errors are listed:



brackets.js

```
47    * Unlike other modules, this one can be accessed without an explicit require() because it exposes
48    * a global object, window.brackets.
49    */
50   define(function (require, exports, module) {
51       "use strict";
52
53       // Load dependent non-module scripts
54       require("widgets/bootstrap-dropdown");
55       require("widgets/bootstrap-modal");
56       require("thirdparty/path-utils/path-utils.min");
57       require("thirdparty/smart-auto-complete/jquery.smart_autocomplete");
58
59       // Load dependent modules
60       var Global                  = require("utils/Global"),
61           AppInit                 = require("utils/AppInit"),
62           LanguageManager         = require("language/LanguageManager"),
63           ProjectManager          = require("project/ProjectManager"),
64           DocumentManager         = require("document/DocumentManager"),
65           EditorManager           = require("editor/EditorManager"),
66           CSSInlineEditor         = require("editor/CSSInlineEditor"),
67           JSUtils                 = require("language/JSUtils")
68           WorkingSetView          = require("project/WorkingSetView"),
69           WorkingSetSort          = require("project/WorkingSetSort"),
70           DocumentCommandHandlers = require("document/DocumentCommandHandlers"),
71           FileViewController      = require("project/FileViewController"),
72           FileSyncManager         = require("project/FileSyncManager"),
73           KeyBindingManager       = require("command/KeyBindingManager"),
74           Commands                = require("command/Commands"),
75           CommandManager          = require("command/CommandManager"),
76           CodeHintManager         = require("editor/CodeHintManager"),
77           JSLintUtils             = require("language/JSLintUtils"),
```

**JSLint Errors**

| | | |
|---|---|---|
| 67 | Expected ';' and instead saw 'WorkingSetView'. | JSUtils = require("language/JSUtils") |
| 68 | Expected 'WorkingSetView' at column 5, not column 9. | WorkingSetView = require("project/WorkingSetView"), |
| 68 | 'WorkingSetView' was used before it was defined. | WorkingSetView = require("project/WorkingSetView"), |
| 68 | Expected ';' and instead saw ','. | WorkingSetView = require("project/WorkingSetView"), |
| 68 | Expected ',' at column 5, not column 68. | WorkingSetView = require("project/WorkingSetView"), |
| 68 | Expected an identifier and instead saw ','. | WorkingSetView = require("project/WorkingSetView"), |

Line 67, Column 62 ★ 319 Lines JavaScript Spaces 4

## Task 1

**Your task is to fix that bug, so that JSLint only operates on JavaScript files** and not on HTML files.

➢ **Where would you make change(s)?**
Please indicate the position(s) for the change(s) by highlighting the line or code snippet and state your change(s) verbally

JSLint is automatically executed when a file is saved. But there is no way to execute JSLint manually without saving a file.

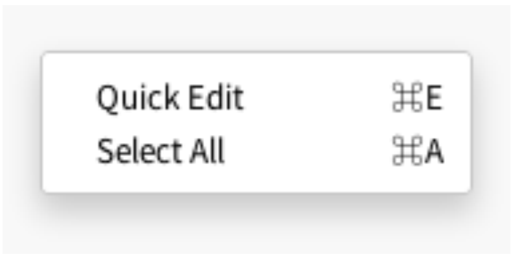
## Task 2

---

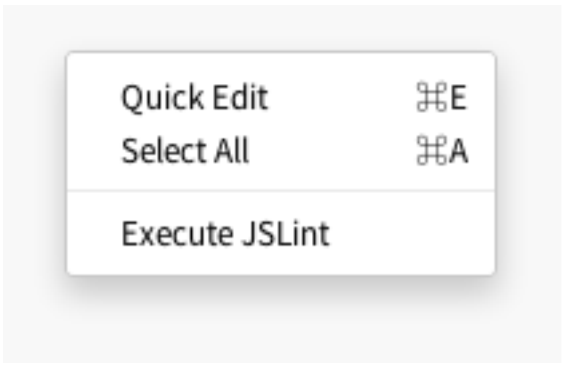
**Your task is to add a new context menu item called “Execute JSLint“ to the context menu of the editor**, **that will execute JSLint** on the currently visible JavaScript file.

- **Where would you make this change or changes?**
  Please indicate the position(s) for the change(s) by highlighting the line or code snippet and state your change(s) verbally

Before:



After:



Info: The **name of the menu item** (“Execute JSLint“) is already given for every language and is available by using

- `Strings.CMD_JSLINT_EXECUTE`

So you don’t have to bother with internationalization.

# Appendix D

# User Study: Provided Sketches

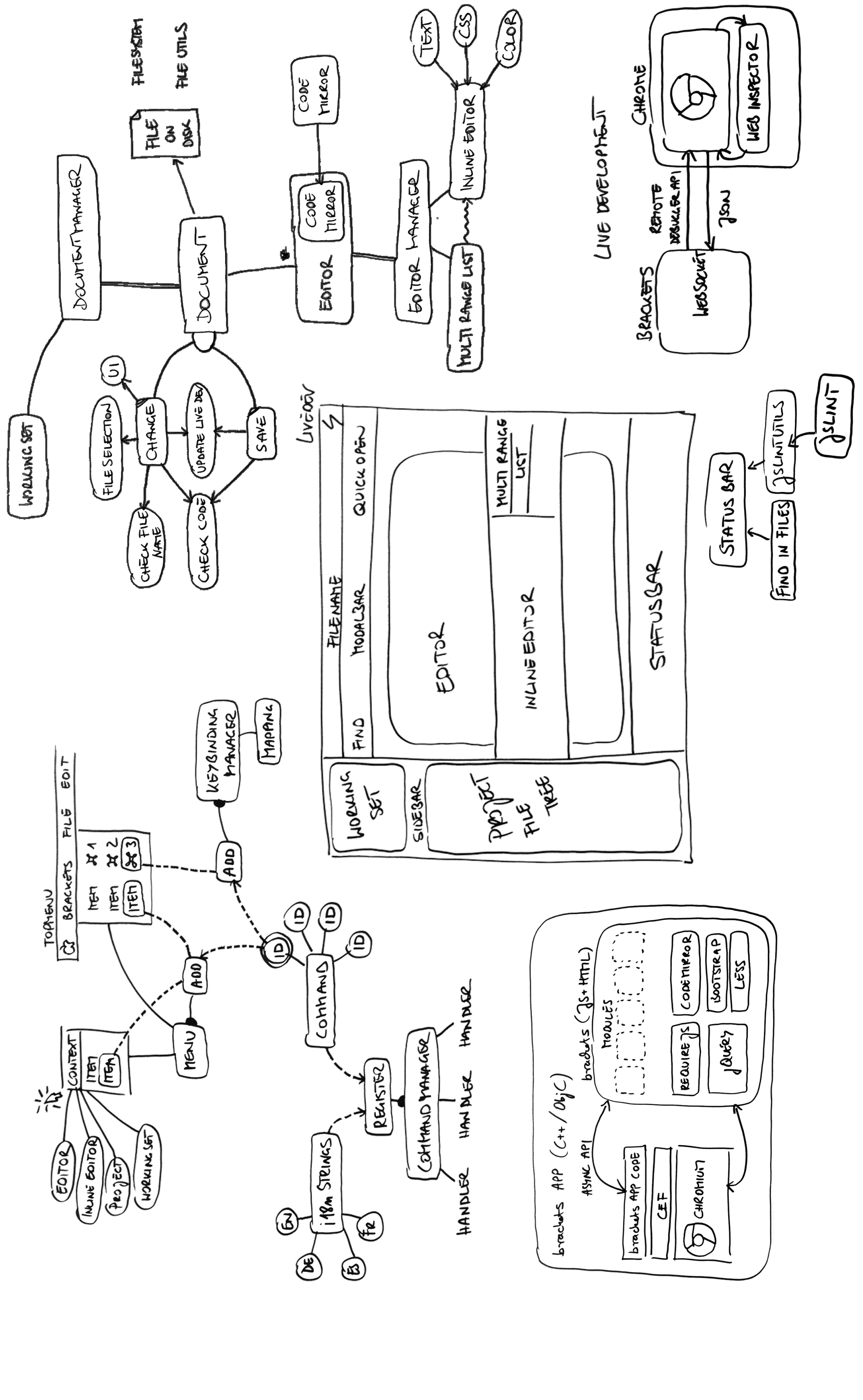


**Figure D.1:** Sketches provided during the user study to both the connection group and the control group.

# Bibliography

P. Baudisch and R. Rosenholtz. Halo: a technique for visualizing off-screen objects. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 481–488. ACM, 2003.

B. B. Bederson and J. D. Hollan. Pad++: a zooming graphical interface for exploring alternate interface physics. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, UIST '94, pages 17–26. ACM, 1994.

B. B. Bederson, J. Meyer, and L. Good. Jazz: an extensible zoomable user interface graphics toolkit in Java. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, UIST '00, pages 171–180. ACM, 2000.

A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola. Code bubbles: a working set-based interface for code understanding and maintenance. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*. ACM, 2010.

S. M. Branham, G. Golovchinsky, S. Carter, and J. T. Biehl. Let's go from the whiteboard: supporting transitions in work through whiteboard capture and reuse. *CHI*, pages 75–84, 2010.

R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543 – 554, 1983.

M. Cherubini, G. Venolia, and R. DeLine. Building an Ecologically valid, Large-scale Diagram to Help Developers Stay Oriented in Their Code. In *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*, pages 157–162, 2007a.

M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM Request Permissions, 2007b.

M. Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial, 1991.

S. P. Davies. Models and theories of programming strategy. *International Journal of Man-Machine Studies*, 39(2): 237–267, 1993.

R DeLine and K Rowan. Code canvas: zooming towards better development environments. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, pages 207–210, 2010.

F. Détienne. *Software Design—Cognitive Aspects*. Springer-Verlag New York, Inc., 2002.

T. Dorta, E. Perez, and A. Lesage. The ideation gap:hybrid tools, design flow and practice. *Design Studies*, 29(2):121–141, 2008.

S. G. Eick, J. L. Steffen, and E. E. Sumner Jr. Seesoft - A Tool for Visualizing Line Oriented Software Statistics. *IEEE Trans. Software Eng.*, 18(11):957–968, 1992.

M. J. Eppler and R. Pfister. *Sketching at work: A Guide to Visual Problem Solving and Communcation*. mcm Institute, 2011a.

M. J. Eppler and R. Pfister. Sketching as a tool for knowledge management: an interdisciplinary literature review on its benefits. In *i-KNOW '11: Proceedings of the 11th International Conference on Knowledge Management and Knowledge Technologies*. ACM, 2011b.

V. Goel. *Sketches of Thought*. MIT Press, 1995.

G. Goldschmidt. The backtalk of self-generated sketches. *Design Issues*, 19(1):72–88, 2003.

J. Good, P. Brna, and R. Cox. *Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension*. University of Edinburgh, 1999.

M. D. Gross and E. Y. Do. Ambiguous intentions: a paper-like interface for creative design. In *Proceedings of the 9th annual ACM symposium on User interface software and technology*, UIST '96, pages 183–192. ACM, 1996.

A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *Software Engineering, IEEE Transactions on*, 32(12):971–987, 2006.

C. Kurtz. Code Gestalt: a software visualization tool for human beings. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 929–934. ACM, 2011a.

C. Kurtz. Code Gestalt: From UML Class Diagrams to Software Landscapes. Diploma thesis, 2011b.

J. A. Landay. SILK: sketching interfaces like krazy. In *CHI '96: Conference companion on Human factors in computing systems: common ground*. ACM Request Permissions, 1996.

T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

T. C. Lethbridge, J. Singer, and A. Forward. How Software Engineers Use Documentation: The State of the Practice. *IEEE Softw.*, 20(6):35–39, 2003.

S. Letovsky and E. Soloway. Delocalized Plans and Program Comprehension. *Software, IEEE*, 3(3):41–49, 1986.

J. Lin, M. W. Newman, J. I. Hong, and J. A. Landay. DENIM: finding a tighter fit between tools and practice for Web site design. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '00, pages 510–517. ACM, 2000.

N. Mangano, A. Baker, M. Dempsey, E. Navarro, and A. van der Hoek. Software design sketching with calico. In *the IEEE/ACM international conference*, pages 23–32. ACM, 2010.

J. Meyer. EtchaPad - disposable sketch based interfaces. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, pages 195–196. ACM, 1996.

H. A. Müller, S. R. Tilley, and K. Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering*, volume 1 of *CASCON '93*, pages 217–226. IBM Press, 1993.

B. Paulson and T. Hammond. PaleoSketch: accurate primitive sketch recognition and beautification. In *Proceedings of the 13th international conference on Intelligent user interfaces*, IUI '08, pages 1–10. ACM, 2008.

N. Pennington. Empirical studies of programmers: second workshop. chapter Comprehension strategies in programming, pages 100–113. Ablex Publishing Corp., 1987.

B. Plimmer and I. Freeman. A toolkit approach to sketched diagram recognition. In *Proceedings of the 21st British HCI Group Annual Conference on People and Computers: HCI...but not as we know it - Volume 1*, BCS-HCI '07, pages 205–213. British Computer Society, 2007.

P. Schmieder, B. Plimmer, and R. Blagojevic. Automatic evaluation of sketch recognizers. In *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling*, SBIM '09, pages 85–92. ACM, 2009.

D. A. Schön. *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, 1983.

K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, 2001.

T. M. Shaft. *The role of application domain knowledge in computer program comprehension and enhancement*. PhD thesis, 1992.

B. Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3):219–238, 1979.

J. Singer, T. C. Lethbridge, N. G. Vinson, and N. Anquetil. An examination of software engineering work practices. *CASCON*, page 21, 1997.

J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *ICSM*, pages 157–166. IEEE, 2009.

B. Tversky. What does drawing reveal about thinking? In *IN*, pages 93–101, 1999.

B. Tversky. What do sketches say about thinking? In *AAAI Spring Symposium on Sketch Understanding*. AAAI Press, 2002.

B. Tversky and M. Suwa. Thinking with sketches. volume 1, pages 75–85. 2009.

A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

J. Walny, J. Haber, M. Dork, J. Sillito, and S. Carpendale. Follow that sketch: Lifecycles of diagrams and sketches in software development. *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop*, 2011.

Y. Wong. Rough and ready prototypes: lessons from graphic design. In *Posters and Short Talks of the 1992 SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 83–84. ACM, 1992.

P. T. Zellweger, J. D. Mackinlay, L. Good, M. Stefik, and P. Baudisch. City lights: contextual views in minimal space. In *CHI '03 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '03, pages 838–839. ACM, 2003.

# Index

Typeset May 21, 2013