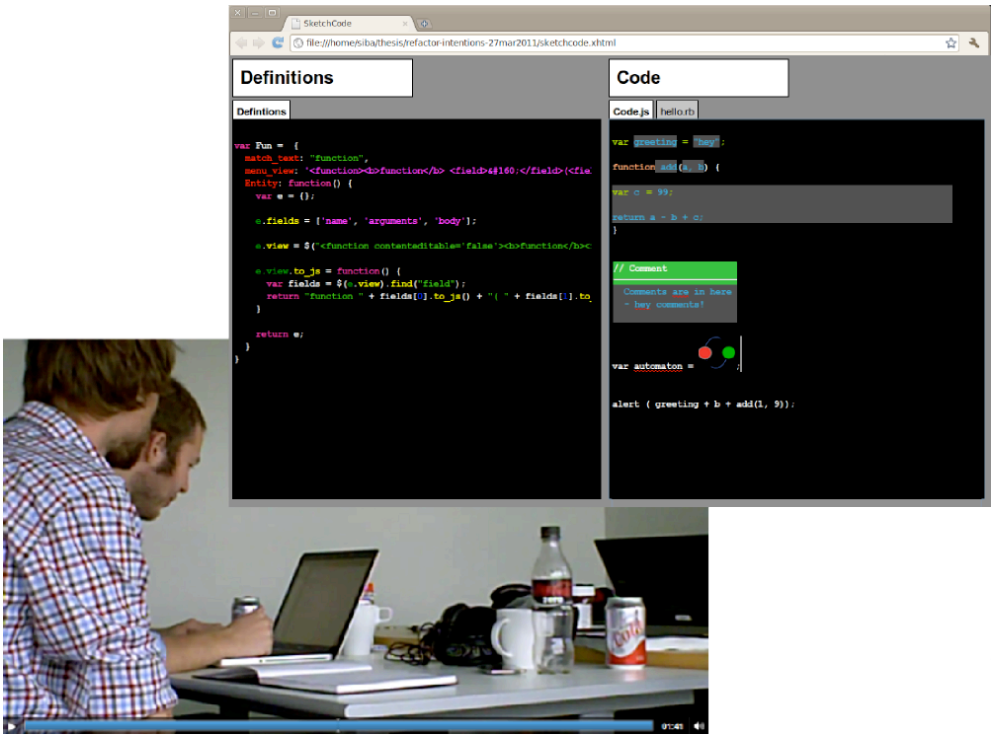
SketchCode

Turning Source Code into a Design Material



Siemen Baader
siemenbaader@gmail.com

Master of Science
Digital Design and Communication
Specialization in User Centered Technology Development

IT University of Copenhagen
August 2011

Supervised by Joseph Kiniry

ABSTRACT

Computer programming is a highly creative and explorative process. In spite of this, current programming tools provide significantly better support for well defined situations than for exploration. This makes programming unnecessarily difficult.

This MSc thesis investigates how design professions use hands-on learning, materials and visual cognition to support creative problem solving. A cognitive model of programming as reflective practice is proposed. Combined with field studies of working programmers, a number of characteristics for programming tools supporting this model are proposed. This includes the ability to interact directly with fragments of code and gain immediate feedback; the ability to express concepts visually; and the ability to devise new representations quickly for new problems as they are encountered.

A design is developed and a technical proof-of-concept prototype implemented as an HTML5 based web application. The design is critiqued by evaluating a number of problem scenarios identified in the user studies and the implementation process of the prototype against the solutions proposed by the design.

It is concluded that the proposed model of programming as reflective practice, and the proposed design, have the potential to simplify programming significantly. It is suggested that future work implements key features of the design and conducts formal usability tests in order to quantify the contribution that programming as reflective practice can make to programming tools.

ACKNOWLEDGEMENTS

I would wholeheartedly thank file-sharing startup Ge.tt for not only letting me study their coding practices extensively, but also helping me with the actual implementation of my prototype. Thanks also to the good folks at the PIT lab, especially Juan Hincapié-Ramos and Aurélien Tabard, for backing my odd-sized ideas so firmly. I am in great debt to professor Irina Shlovski, who gave me the room and encouragement to pursue my own research interests early in my studies at ITU. Last but not least, I would like to thank my supervisor, Joe Kiniry, for his exceptional support and mentorship at all levels of the project, be it from proofreading my non-native English when asked to, over personal productivity, to putting me in contact with the major industries in this field.

Dedicated to the loving memory of Why the Lucky Stiff.
His spirit will go on among us.

CONTENTS

1

INTRODUCTION, PROBLEM AND DESIGN VISION

1.1 SOFTWARE DEVELOPMENT IS AN EXPLORATIVE PROCESS

Over the past few years, agile practices have gained considerable popularity. They are, at least in some areas, increasingly replacing more plan-and-execute-oriented methods derived from physical construction processes. Agile methods accept that software development is more complex than most traditional production processes. It is characterized by uncertainties both with regards to discovering the exact requirements (which often only emerge when real users are confronted with the system), and at the implementation level (where implementation plans frequently turn out not not work as hoped for). In a plan-driven process, these uncertainties tend to accumulate.

Agile practices deal with them by focusing on constant evaluation and adjustment instead of precise planning. They break projects down into increments, each resulting in a release integrated with the previous ones. In this way, each iteration resolves its own uncertainties, and adjustments to the plan are made for the coming iterations if needed.

Agile practices support this process at three distinct levels.

- At the **project management and requirements level**, agile processes deliver a working increment of software after each iteration. This increment can then be evaluated by business experts, or by being put to real use. If it for some reason is not useful, alterations, new features or even cancellation of the project can be decided. Agile project management has been popularized especially through Scrum[Schwaber and Beedle, 2001].

- At the **software engineering level**, agile practices provide techniques to keep software in a malleable condition. Extreme Programming [Beck, 1999] provides engineering practices that allow a system to be modified and kept in a working state at low cost. Automating testing allow programmers to change the system without fearing to break it. Co-located teams and pair programming ensure that the right knowledge is available to competently make changes.

- At the third level, agile practices are supported by **development tools**, which aid rapid evaluation and modification of the software. Revision control systems allow teams to keep track of changes made by many developers at the same time, and unit testing tools and continuous integration systems automatically check the system for defects. Refactoring tools assist in changing the structure of programs efficiently. While most agile literature is deliberately tool-agnostic, the origins of agile can be traced back to dynamic language communities, especially Smalltalk. Today, dynamic languages as PHP, Ruby, Python and Javascript are gaining terrain, arguably because they support a more flexible process than compiled languages.

1.2 TOOLS ARE NOT AGILE ENOUGH YET

By embracing the agile paradigm, we have accepted a basic premise about the nature of software development: That it is a complex and explorative process, where the exact path is always unknown. In it, we are at all levels better off by systematically acting, learning and reacting than by thinking out one grandiose plan in advance.

If we look at today's tools, support for agile development appears to have increased over the past ten years. Especially on the web, Dynamic, lightweight languages such as Python, Ruby and server-side Javascript are gaining popularity at the expense of more complex, compiled languages. They typically require less code to be written for the same functionality, and allow execution without compilation, resulting in a shorter modification cycle that aids the process.

Yet, if we look at these tools from a design theory perspective, there is great room for improvement.

The exploration of a complex solution space has traditionally been the subject of the design professions. Superimposing design theory on today's agile tooling exposes four concrete inadequacies:

Interactive Manipulation is Limited

Design situations are so complex that they cannot be thought out by mental reasoning alone. Instead, designers use an interactive process of sketching techniques and materials to externalize their ideas, reflect, and learn from them. This process is sometimes referred to as 'thinking with your hands'.

Programs typically have only one entry point. Whenever a programmer works with a part of the program which is far away from that entry point, the program must be restarted and brought into the appropriate state (either manually or using a test fixture) before the outcome of the change can be observed. This both interrupts the mental flow of exploring by forcing the programmer to provide the setup, and consumes time.

This favors mental reasoning over acting and observing consequences, and reduces the number of approaches a programmer can feasibly explore empirically.

Concepts are not Expressed Directly

Central to design processes is the frequent re-interpretation of a situation into a concept which then provides a framing for thinking and acts as a generative metaphor. In addition to mental framing, using a concept also allows designers to sketch and prototype representations at the level of the concept, omitting detail irrelevant at the moment.

Software development also makes heavy use of metaphors such as design patterns or finite state automata, but also more informal metaphors such as thinking about an object as 'doing' something in 'collaboration' with other objects.

Today's programming tools rely on textual code as the only powerful representation. Other representations are often cumbersome to edit, integrate poorly with each other, and are in general mainly used to ultimately generate text code. In addition, they typically only work for pre-made concepts (think GUI builders) and can only represent off-the-shelf concepts. Supporting a newly identified concept requires new tools to be made (think e.g. Domain Specific Languages), which are projects in their own right.

This forces programmers to mentally bridge the gap between the concept and the code. As any manual process, this is error-prone and slow, and a distraction from the conceptual problem. In addition, it deprives programmers of the ability to support their reasoning by using the right visual representations, putting them in a similar situation to a mathematician who is prohibited to draw any charts. Finally, because editors have no conceptual understanding of the problem being solved, they cannot assist with context-specific input methods, or by automating some of the symbolic reasoning about the concepts.

Exploration Support is Limited

Design situations regularly deal with several partially conflicting demands at once. These are solved by lateral problem solving approaches - by treating the individual requirements as unclear. They are then re-defined to reach a situation where these demands eventually are all met, but in different ways than perhaps initially anticipated.

This requires a process that can quickly propose several alternative scenarios and compare them, resembling a breadth-first search. Designers use sketches, prototypes and materials which are quick to make and only exhibit the properties relevant for the particular problem to explore these alternatives.

Programmers also frequently face situations where one implementation can be substituted for a different one which practically fulfills the same need, but which is much simpler to implement. Most often, this simpler way only becomes clear after a more complex approach has been attempted.

Tool support for exploring alternatives lies mainly in branching revision control systems, and to lesser extend in editors undo histories, or working on spike solutions stored in separate folders. Programmes can use them to switch between different states of their program and compare them. This limited support is problematic because it again requires developers to explicitly deal with the bookkeeping task of storing the specific situations. For instance revision control systems typically refuse to store revisions without a commit message, and are thus more oriented around a conscious archiving metaphor than unobtrusively capturing design situations for later comparison.

The Most Suitable Tools are too Strange for the Mainstream

Identification of the deficiencies here is guided by design theory, but they over time also have become clear through practical experience to many programmers; especially within particular enthusiastic hacker communities. In these niches, tools have existed for decades which in several ways address these issues.

Within the Lisp community there is a long tradition for using macros, which in essence re-define the language itself to express any concept encountered. Emacs, the de-facto Lisp development environment, features both direct evaluation and interaction with any part of a program at hand (under cursor), and an extension mechanism which

is heavily used to provide editing support for any concept, including calendars and todo-lists.

Smalltalk provides a complete system that can be extended while running. It includes a graphical environment (Morphic) that at the same time can be used like a traditional GUI, and easily modified, allowing a very interactive GUI building workflow.

However, these environments have never gained mainstream adoption. This is arguably because they favored their own superior designs over pragmatic adoption to mainstream use cases. They either did not integrate with the existing practices of their intended users. Smalltalk runs from a (large) binary image, not source text files, which did not integrate well with file-based host systems. Lisp is file-based, but on the other hand has a syntax consisting of lots of parenthesis which looks nowhere like anything users of most languages have seen before.

These unfamiliarities confront new users with steep learning curves while, at the same time, the advantages of these technologies and their more powerful concepts are hard to assess for new users.

1.3 DESIGN VISION

The vision of SketchCode is it to provide a programming environment that supports agile web development for the mainstream, in way similar to how design materials support the work process of designers.

2

EPISTEMOLOGY AND METHOD

Overlooking the field of programming tools at large, one gains the impression that most tools have been built out of self-interest by specific communities, to support their own practices. Computer Science brought Lisp, Emacs and functional languages, Software Engineering visual IDEs like Eclipse, Human Computer Interaction 'easy' systems like Scratch[1] and Hands[2], and Design brought Processing[3] and Field[4]. These tools support the specific problems that their communities deal with, but – more interestingly – they also reflect the thinking and cultural assumption about programming as an activity of these communities.

SketchCode is a little different from these tools in that it tries to support one community – that of agile web hackers – with the thinking from another – that of Design. At the same time, it has been designed within the context of academia, on a design master program, but in close everyday contact with a Computer Science Department. This cultural intermingling provides great opportunities for new approaches, but also the risk of misunderstandings based on different cultural assumptions.

To deal with this, I will in the following dissect the underlying philosophies – the epistemologies – of science, engineering and design. They are discussed with regard to the specific problem of designing SketchCode. Based on this, I present how I combine aspects from design and academic traditions as the epistemology of this design process. At the same time, these sections serve the purpose of filling readers from different backgrounds in on the respective other fields. This allows me in the rest of the report to take these basics for granted and just refer back to them – e.g. when I explain how design practices map to programming activities.

2.1 SCIENCE

The aim of science is it to find true, generalizable explanations to existing phenomena in the world around us[Cross, 2007]. A scientific

1 http://scratch.mit.edu/
2 http://www.cs.cmu.edu/~pane/research.html
3 http://processing.org/
4 http://openendedgroup.com/field

problem is not considered solved before an explanation is found that can fully describe the observed phenomenon. This results in very reliable findings, which ultimately can be applied to predict the world around us. They can also form the basis for other research, which do not need to test accepted findings again. In this way science has an accumulative nature; it constantly expands the range of phenomena that we can reliably predict.

The scientific approach is problem-centered, in that its most prominent tool to focus understanding is the research question. The research question specifies precisely which phenomenon to investigate. Only a sharp, well defined question helps defining hypotheses, experiments to test them, and allows the researcher to collect only the data relevant to the research question from an otherwise data-wise overwhelming environment. Re-focusing the research question is a fundamental trait of science, and allows researchers to establish this level of thoroughness, at the expense of breath. Thus, science is also reductionistic in nature.

The fundamental understanding of how to learn about the world - that is, its epistemology - is positivism[Schön, 1983]. It takes a rather conservative and humble approach to what can be believed as true. It accepts only what can be - positively - confirmed by using our senses by repeated empirical demonstration. Less rigidly confirmed knowledge, such as personal experience, untested ideas or intuitions or anecdotal evidence, are of course relied on in the daily work of scientists, but ultimately not accepted into the body of knowledge.

The bottom line is that information is distrusted unless proven true.

The aim of expanding the body of knowledge is reflected in the practices of the scientific community. Because uncertainties in the body of knowledge invalidate its value, scientist put great efforts into transparency of their methods, and the indicating which knowledge they build on through references. This ideally allows other scientists to verify every step, which is important, because any invalid assumption can invalidate findings that rely on them.

With regards to making new systems, the downside of this extremely thorough process is that it progresses rather slowly, that it stays away from complex problems, and that it has no tools for generating and selecting among new approaches.

2.2 ENGINEERING

Engineering disciplines draw on the body of knowledge established by the sciences, and apply it to problems in society, ultimately to produce

value. Engineering is often technical, but also usability engineering is an examples of putting to use the predictive powers of scientific understanding.

Engineers typically work on well-defined problems that are known to be conceptually possible, but not trivially done[Schön, 1983]. They apply their specific domain knowledge, tools and processes in a relatively linear progression, ultimately solving the problem. This often makes engineering take the shape of an in-depth, optimizing process.

Like design and science, engineering does involve creative thinking and sifting of alternatives. However, engineering does happen within the boundaries of a defined problem that is known to be solvable by engineering.

The dominating epistemology of engineering is what Schön denotes Technical rationality. Coming from applied research, engineering shares a problem-centric view, and assumes that, once a problem is defined sufficiently clear, it can be solved by rationally applying the knowledge and methods established by the sciences.

Like the sciences, it deals with uncertainties by experimental hypothesis testing, often in the form of the construction and testing of prototypes.

As an approach to creating new systems, engineering is strong when it comes to implementing and testing the understood, but tries to avoid dealing with the unknown. In an explorative design process, it can be a strong driver because implementing and testing provides grounds for making decisions, but it must be complemented by other activities that frame problems for engineering to solve.

2.3 DESIGN

Unlike science and engineering, design does not seek to find truth about something that exists, or to solve a well-defined problem. Instead, it is concerned with the creation of a desired, yet broadly defined future situation [Cross, 2007].

To achieve its goals, design needs to deal with the complexity of the current and the new desired situation. Science and Engineering remove complexity by focusing on individual problems. Designers instead use methods and processes to abstract away for-the-moment irrelevant detail, while retaining the complexity of the whole situation. While the two former disciplines are depth-first activities, design may be seen as a breadth-first search for solutions[Buxton, 2007].

Exactly what information is relevant in a given design situation changes rapidly during the course of designing. It cannot practically be conducted in a top-down manner, dealing with the most abstract levels first, and then working out the more detailed levels, because high-level decisions (e.g. the external structure of a building) impact on the lower levels (e.g. the distribution of sunlight in the interior). Therefore designers work at many levels of abstractions and make broad use of illustration techniques and physical and digital materials to quickly judge the properties relevant at the moment.

This heavy use of materials and the hands-on approach - known as sketching or, more generally, 'thinking with your hands' - both answers question about articulated questions about design ideas, e.g. spatial configurations, but it also inspires new ways of seeing a problem and not thought of possibilities. This leads to the simultaneous development of solution and problem understanding. This is an essential trait of design practice, and a way of dealing with the complexity of design problems. It has been shown that designers who attempt a more problem-focused approach and tried to clearly define a problem statement underperformed[Cross, 2007].

The frequent re-interpretation of situations is also reflected in the fact that designers tend to treat all problems as ill-defined; even initial requirements. This allows them to reconfigure a design by removing functionality from a design and replacing it not with a similar feature, but an entirely different, yet adequate, configuration. This contributes to the breath-first nature and allows the designer to continue to add to the design even when he has to take into account the consequences of previous design decisions.

This explorative and reinterpreting process has been illustrated by Schön [1983] as a cycle of seeing-moving-seeing. Decisions are made based on reflections (seeing), i.e. situated learning, and the epistemology of the design professions (which Schön argues is present, but rarely acknowledged, in any practicing profession) is denoted by Schön as Reflective Practice.

Reflective Practice has specific needs for information to base it decisions on. It requires quick and easy access to rich information about both requirements, users and use scenarios, and consequences of design decisions. On the other hand, decisions made become part of the (current state of) the design and are subsequently tested, altered or even removed again by interacting with the rest of the design during the design process. This allows designers to work with less reliable, but quicker or richer data.

Modern design makes use of qualitative methods such as open ended interviews and ethnographic(ally inspired) field work for user studies,

but also informal testing of products by the designer using them himself. Arguably some of the best designs come from designers who are part of a given domain, and who possess rich, but subjective, first-hand experience with it - knowledge that cannot efficiently be conveyed through desk research or field reports. Other techniques for receiving quick, rich data include using well-informed people (either experts or users) as efficient sources, and frequently confronting prospect users with design proposals.

2.4 IT, DESIGN AND ACADEMIA

Several authors (e.g. Buxton [2007], Löwgren [2007]) have argued that many IT systems and products fail to deliver value, not because of technically inadequacies, but because they were solving the wrong problems. They were unable to serve their users because they were specified either through the informal intuition of engineers, or extracted by scientists investigating the current practices, which manifested the need for a new system; not its specification. These authors argue for adoption of a design perspective in the specification and construction of systems. This should provide tools and methods to evolve the right specification in dialog with the intended users.

Initiatives rooted in the design professions like e.g. the Copenhagen Institute for Interaction Design[5] are now embracing IT as a first-class design material. Typically, design institutions focus more on production and practice than academic qualities.[6] At the same time there is a shift of the traditional design schools, e.g. the Danish Design School, in moving towards an academic system, issuing research-based master and Ph.D. degrees. The ITU, likewise embracing the combination of design practices and IT, comes from an academic tradition.

As the previous sections have illustrated, there are conflicting elements in the cultures of design and science, and adopting design practices within an academic context requires being explicit about what aspects of science and design to adopt.

In the following, I will therefore make explicit which fundamental beliefs and practices from the three professions of Science, Engineering and Design I subscribe to, which ones I have to reject, and how I frame them within an academic context.

5 http://ciid.dk

6 Quoting one CIID official: "We want our students to make awesome designs; not to write papers."

From the the presented traditions, I adopt the use of established theories, transparency, guidance by a design vision instead of a research question, dealing with complexity by using abstractions rather than singling out a problem, accepting rich but less rigid data, and the process of reflective practice.

As customary within academic traditions, the project makes heavy use of established theories to guide the design. This allows it to make better predictions and quickly assess the potential of different alternatives than less theoretically informed, more hands-on approaches. I also attempt to be academically credible by being transparent about the process, and to state to what degree I am certain of what I present - i.e. whether decisions are based on credible data, plausible explanations, or just personal experience.

I reject the problem-focused approach because of its unsuitability to deal with the complexity of a design project. It is replaced by the design vision - the goal of SketchCode, stated above. Complexity is dealt with in a designerly manner, by using appropriate representations and materials for the situation, and by prioritizing sub-projects against the design vision - not by isolating one aspect.

Compared to a traditional academic project, I accept less reliable data to base design decisions on. These are data gathered from qualitative field studies, but also personal experiences, questioning friends and coworkers, and participating in conferences and meetups. While I would not perceive this kind of data as 'true' in a scientific sense, I have accepted them as reliable enough to let them propel the design process. This is possible for three reasons. first, the cyclic design process revisits decisions as discussed, and in effect tests earlier decisions. Second, design is not an optimizing process, and also only partly improvement is better than none. And third, I am concluding the project by testing the final design by evaluating it qualitatively against established theory.

Process-wise, having rejected to define a research question up-front, I rely on reflective practice, like design does. I subscribe to the notion that knowledge about the desired design can only emerge from proposing solution attempts and reflecting on them, not from analysis or desk research alone. This applies equally well to the choice of theories, which only emerge gradually as a way to explain the current understanding of the design situation.

A side effect of this is that the traditional hypothesis testing structure often applied in academia is not easily transferred to a project like

this. The cyclic nature of reflective practice constantly closes the gap between hypotheses and reality, and while identifying that a given theory is not valid is a valuable contribution to science, it is only a step in the process of designing.

Instead, a project like this will present the reflections that occurred during the process, and the didactic reflections on the project itself.

2.6 METHOD

The literature on agile software development, design processes and programming environments is studied, and it is discussed how the three fields relate to support the programming process.

Based on this, a software prototype of the programming environment is implemented as a local web application, using HTML and Javascript. The programming problems that occur during this process are analyzed and solutions proposed according to the informing theories.

In this way the running prototype serves the dual purposes of providing concrete use cases to design around, and of exploring feasible implementation strategies for a programming environment using HTML5.

Alongside the application prototype, sketches and paper prototypes are developed. These serve the purpose of exploring the design space further than what would be feasible with a software prototype alone. Together with the software prototype, they describe the full design.

To ensure that the designed system is useful for its intended user group, a user centered design approach is taken. A group of professional web developers is permanently associated with the project, and interviewed and observed. They are regularly confronted with the prototypes. This process should ensure that the designed environment both fulfills the needs of its intended users, and that it integrates well enough with concepts familiar to them in order to allow wide adoption.

A participatory design process is not chosen because the design relies heavily on theoretical understanding of agile methods, theory on how designers work, and various existing programming tools. Practitioners in the intended audience typically do not have this knowledge. Instead I as the designer seek immersion in the practices of the users. The obvious risk in this – that of designing a system which works for the designer, but which is hardly learnable by others – is addressed by the user centered process.

The end result of this process will be the specification of a system ready to support agile web development in a new, but effective way. All major questions regarding its usefulness, technical feasibility and chances of user adoption will be addressed, and the design will be ready for software engineering. It is not intended to develop stable or useful application software in this project.

The design is evaluated against its goals by a qualitative discussion of how well the mapping of design process theory in practice could be applied to the development of the software prototype and how well the concrete problem scenarios during development could be addressed by the proposed design.

3

SKETCHCODE PRESENTED

This chapter presents the reader with an overview of the concrete design that emerged from the design process. The remaining chapters address specific aspects and the process individually.

The design proposes four main features that support program development as reflective practice. First, it allows any piece of code to be run directly in a local ad-hoc environment to support quick exploration of ideas in-place. Second, it tracks changes in the background as the programmer works with the application, and allows him to quickly access previous attempts. Third, it allows the programmer to represent structures of the program using custom HTML representations to work with the program at the appropriate conceptual level.

3.1 DIRECT MANIPULATION

SketchCode supports direct manipulation by allowing the programmer to run and interact with any section of a program directly.

The programmer does this by making a selection, and pressing Ctrl+Enter. SketchCode then goes into Interactive Mode and displays a setup area above the code, and an interactive console (the Read-Eval-Print-Loop, REPL) below it. The setup area is used for optional code that the section depends on. The code interacted with is denoted a Code Section.

When CTRL+Enter is pressed in interaction mode, the setup code is evaluated, then the section, and finally, the REPL is given focus. The programmer then uses it to play around with the section. He may revise the Code Section, and press Ctrl+Enter to restart the scenario.

The REPL provides checkboxes for each statement that is evaluated. When these are checked, it remembers the statements across restarts. This helps the programmer to define expected outcomes as they become clear, and gradually beat the code into shape.

To assist this, the REPL provides assert() and log() methods that are available to the Code Section, setup, and REPL contexts. They may be used to inspect the state of variables anywhere in the code. log()

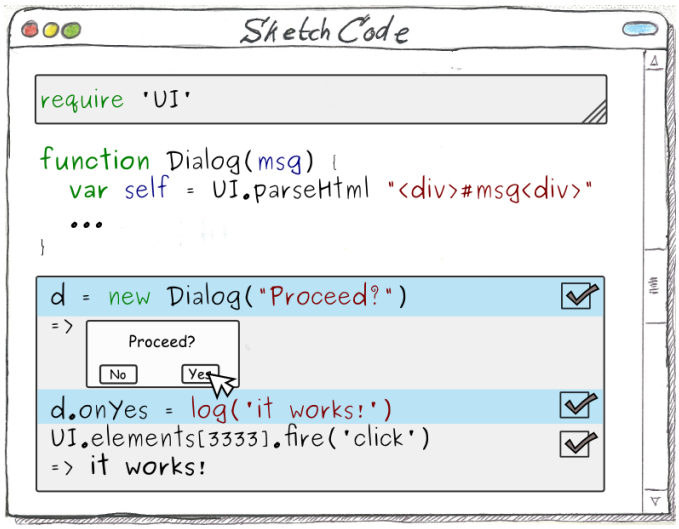


Figure 1: A dialog UI element being tinkered with using Interactive Mode

simply logs its arguments to the REPL output, while assert() accepts an expression. If the expression is false or raises an exception, the REPL will show it with a red background. This makes it easy to detect unintended behaviour.

The REPL readily displays DOM elements. This allows the programmer to interact not only with sections that have to be called with code, but also UI elements and mouse event handlers. By default, SketchCode logs mouse events when CTRL is pressed and inserts a statement into the REPL log that can be checked and kept to reproduce the mouse event automatically when the scenario is restarted. By associating assert statements with event handlers, the programmer can use this feature learn and define, and eventually automatically test the behaviour of UI elements.

The interactive mode has similarities with common unit testing practices, that also run and test pieces of code in isolation, but is not meant to replace them. Instead, it provides a simple and direct way of interacting with code, form a conceptual model of it, and change it into the desired shape.

The interactive mode can be left by pressing Escape. Outside interactive mode, the assert() and log() statements remain, but are hidden and return immediately with no effect. Leaving interactive mode leaves a small label at the beginning of the selection. The interactive mode can be restored by double-clicking the marker, or by moving the cursor into the label and pressing CTRL-Enter. The setup code and REPL log are retained. New interactive modes, covering the same section, but

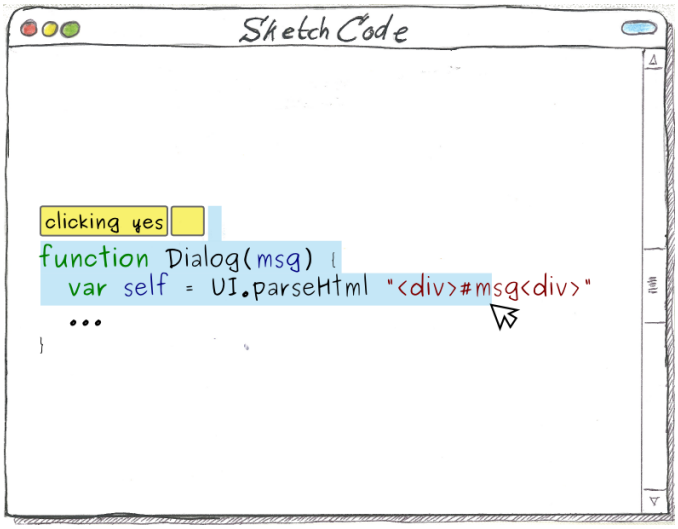


Figure 2: Two markers indicating interactive modes, one labelled. Code is selected to create another session.

probably different concepts, can be added by selecting the code again. Their markers will show up next to the previous marker.

Finally, saved interactive modes and their associated setup code, REPL history and logging statements can be deleted by deleting their markers. They may also be given a label by placing the caret inside the marker and typing.

Interactive Mode is intended to benefit programmers by providing them with more, and better qualified, information about their program and their design options available. It does this in the following ways:

Running code sections directly allows ideas to be explored within the original context. This helps the programmer maintain a clear picture of the surroundings, but on the other hand allows him to develop the approach without having to pay too much attention to the problem of integrating it before it is done. Integration can be done piecemeal by gradually referencing all conflicting components in the setup code and integrating them.

The unobtrusive collection of statements and intended outcomes by the REPL result in code that captures the design intentions of components. They can be replayed, and possibly refactored into a unit test. This provides similar protection against regressions as the unit test produced by the (more plan-driven and hard to master) practice of Test-First Development. But it does not compromise on the explorative, incremental style of working, where ideas are developed gradually by trying them out quickly.

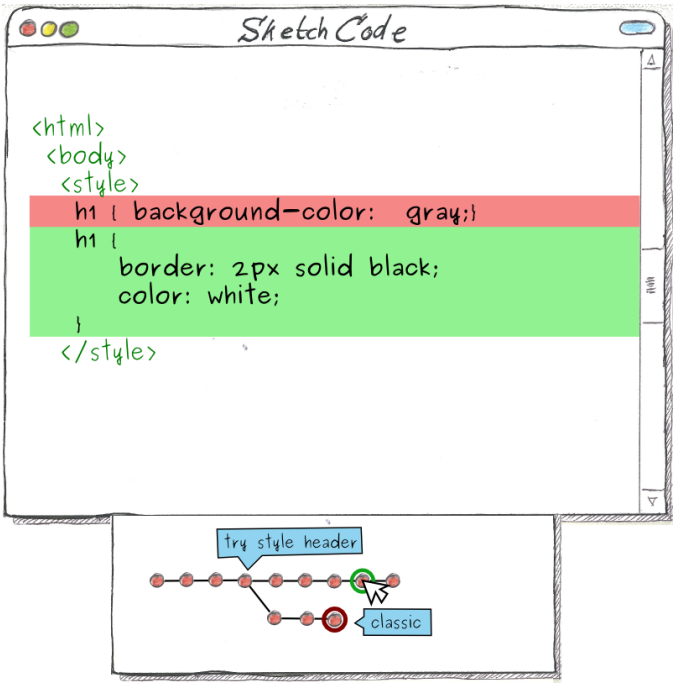


Figure 3: Tuning CSS styles for a heading with explored alternatives ready-at-hand.

3.2 UNOBTRUSIVE EXPLORATION HISTORY

SketchCode supports exploring alternatives it by combining its undo history with revision control, unobtrusively tracking the state of all files in the project each time the program is executed, and allowing the user to work with the history as an integrated part of development.

Whenever the user runs a piece, or all, of his program by pressing Ctrl+Enter, the current state of all files in the project is recorded, and silently committed to a branching revision control system. Pressing Ctrl+Z and releasing it restores their state to the last commit. In effect, this moves back in the history. Ctrl+Shift+Z moves forward along the path the user came, ultimately reaching the uncommitted changes. In this way, the feature works as a coarse-grained undo history.

Holding Ctrl+Z reveals a tree view of the history that remains visible as long as the keys are held pressed down. In this view, the user can use the arrow keys to move a pointer from the current checkout to any other node. As nodes are selected, or hovered with the mouse, the editor shows a diff view of the current state, and the hovered one.

The user checks out a commit by pressing Enter or double-clicking on it, which loads all files in the editor to the state they were in at the

time of the commit. Working from here automatically creates a new branch.

Commits by default happen silently and do not require a commit message, but the user may add them later to help him overlook the revision tree. This is done by right-clicking on a commit, selecting an 'add label' command from a pop-up menu and entering a name.

In trivial situations, the user would use Interactive Mode and the Exploration History by simply trying several approaches departing from one commit, and then continuing from the branch of the most desired solution. In practice, past approaches are often reconsidered and found practical in new situations, many commits ahead. The easiest way of inserting these into the current branch is simply visiting the old commit, and copying the desired sections. RCSes provide powerful merging tools that allow cherry-picking sections and preserve the partial merge in their history. Since SketchCode uses an RCS as its back-end, this is possible, but copying often does the job, and keeping a precise history of the project itself is less important than inserting the right feature and moving on.

The History is a complementary feature to Interactive Mode, and, in the same way, automates some of the bookkeeping of exploring different approaches. It allows the programmer to stay in the creative loop of exploring and making decisions. The feature does not differ greatly from traditional use of branching revision control systems, but is applicable ready-at-hand also at a smaller scale, and is less intrusive.

3.3 INTENTIONALITY AND CONCEPTS

The Interactive Mode and History features make it easy for the programmer to try out different approaches, interact with them, and clear his mind about which one is best. The other side of the coin is refining a solution to a clean, meaningful shape, once it is found. Technically, this encapsulates all unnecessary detail, and allows the programmer to only deal with the relevant parts when re-using it. On the conceptual level, it also separates what a piece of code is supposed to do - its Intention - from how it does it - its implementation. Clear intentions allow the programmer to form precise conceptual models of program components, what they can do and what they need. In the explorative cycle of trial, error, and correction, a better understanding of components helps the programmer make better tries in the first place, and avoid spending feedback cycles on detecting trivial misuse of components. Vice versa, if the programming environment understands the intention behind a component, and not only the source

code of its implementation (footnote: akin the spirit of the law vs its letter), it can aid with intention-specific assistance and guidance.

SketchCode encourages the representation of higher level concepts directly in the editor through Concepts, which are sections of code that are represented using an arbitrary rich editing view defined in HTML, and compile to corresponding source code.

3.3.1 *Using Concepts*

A Concept may be any section of code that has conceptual meaning to the programmer, and that can be expressed in code, but may be shown in a better way than code.

To use a concept, the programmer either right-clicks in the source code and chooses the desired concept from a pop up menu. Concepts have names, and they may also (more conveniently) be inserted through an autocompletion menu, if the programmer types their name, and selects the concept from the menu.

When a concept is used, a concept instance is inserted into the code. It is shown as a rich editor, with appropriate parameters to specify. In the example, the editor is a color wheel, and the parameter the actual value of the color - its CSS hex code. It is up to the concept's editor to provide the appropriate means to choose it. This particular concept editor offers selecting the color visually on the color wheel, or entering it directly using the keyboard.

Before code is evaluated, all concept instances are rendered to code. In this case, the color wheel is rendered to the hex code, just as if it were entered directly in plain text.

3.3.2 *Evolving Concepts*

SketchCode allows programmers to define and revise concept editors gradually, as a part of the normal refactoring process, using the same Web UI technologies that they are using in their own projects. This is done using HTML, JavaScript and CSS based UIs, including the options of mathematical editors using MathML, or visual ones using SVG or Canvas elements.

To define a concept, the programmer opens a special file named Concepts, which is part of the current SketchCode project. Here he

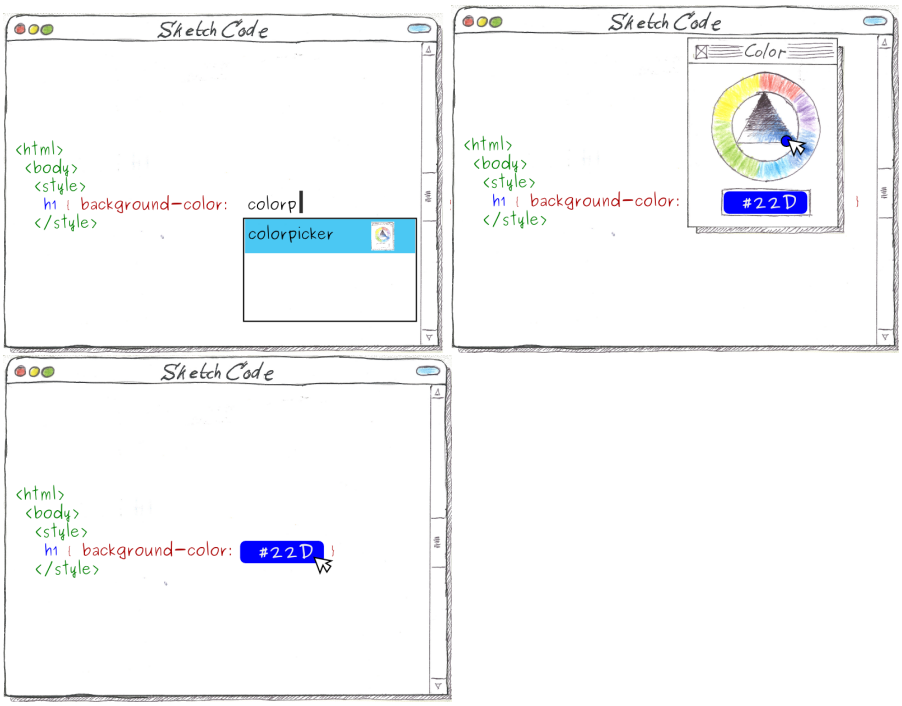


Figure 4: A CSS color concept is inserted, used to select a color, and stays inlined with the code.

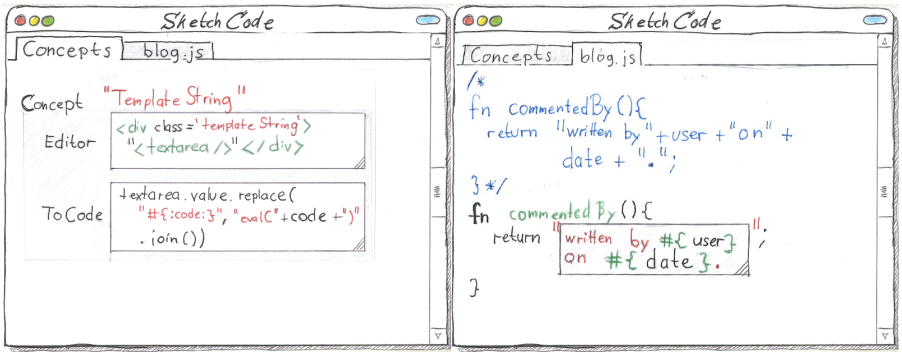


Figure 5: A concept is defined and subsequently used to refactor code to a higher level of abstraction.

inserts an instance of concept definition, which is itself a built-in concept.

Apart from a name, the concept definition editor requires two important properties to be defined.

The Editor property defines the look and behavior of the editor as a UI element, using HTML, JavaScript and CSS styling. The purpose of the editor is it to provide the most appropriate editing support for the concept instance, as illustrated above. Like any rich HTML UI widget, it may keep its own logic and data models.

The To Code property contains a script that controls how the concept instance is converted to plain source code, before the program is evaluated. The script would typically extract data from input fields in the editor, transform it, convert it to text, and output it in the form of evaluable JavaScript. The Concepts file is special in that it is read and evaluated on each start, and can be re-run manually after each change to it. The concepts defined in it become then available to the SketchCode editor.

For an example, consider a concept such as a templating multiline string, which some languages have, but others, including JavaScript, have not. Such a string would immediately when it is declared evaluate expressions within special markers (e.g. #{ expression }), cast them to strings and concatenate them with the surrounding string fragments.

To extend his current SketchCode project to allow him use this kind of strings, a programmer creates a new concept, and defines an editor using a simple textarea element. The To Code property would be a function that, when run, takes the content of the textarea, splits it into string fragments and expressions, and returns a string containing a valid JavaScript expression that concatenates the string fragments and expressions.

First class concepts in SketchCode, such as the Code Section concept in Interactive Mode, and concept definitions, are built-in concepts themselves.

4

WORKING PROTOTYPES

A driving force in the design of SketchCode has been the implementation of several crude running prototypes. They had the dual purpose of exploring how the proposed features can be implemented in a web browser, and bringing the design forward by detailing technical possibilities that might be used to design features from.

This section presents the two major prototypes, and discusses how the proposed features can be implemented on a technical level.

4.1 INTEGRATED PROTOTYPE

Throughout the process, several aspects of the described design were prototyped. With the exception of Interactive Mode, they were integrated into this first prototype – hence its name.

In the definitions panel on the left, a concept to accommodate functions is defined. Whenever the definitions panel is focused, and the user presses Ctrl+Enter, the prototype evaluates the content of the panel. Any concepts defined here are then known to the editors, and may be used in any of the panels.

The concept function concept is instantiated in the Code panel in the center, along with a concept for declaring and assigning variables. These particular concept editors show the concepts with colorful styled HTML, but otherwise with the same usual syntax. The text editing panels themselves do not highlight any keywords, and the gray code is plain text.

The lowest line in the Code panel shows how concept instances are inserted using the autocompletion menu. The user has just typed 'var', and the menu suggest either to insert an instance of the (variable) Declaration and Assignment concept, or the plain text 'var' (which admittedly is more useful for longer text fragments to be autocompleted). Concept definitions set the name by which they are recognized trough their match_text attribute as shown on the left side.

The Interactive panel on the right side is available as an output area, and is accessed as a DOM element trough the reference 'app'. The

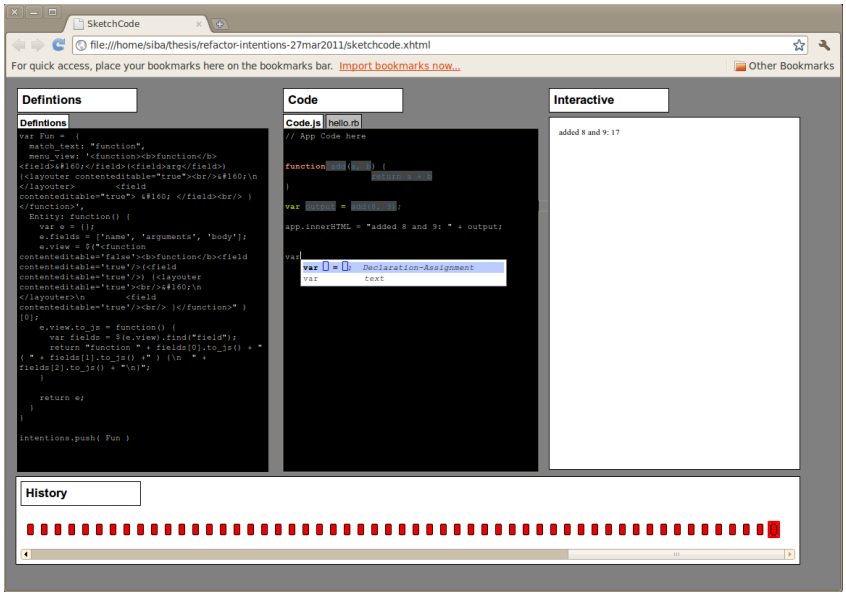


Figure 6: The integrated prototype showing a concept definition, concept instances, concept insertion, history and a live application view.

sample code uses it to display the result of a calculation, but it may also hold UI elements made from DOM elements.

When the code panel is focused, and the user presses Ctrl + Enter, first the definitions panel es evaluated, and then the code panel. The content of the panels is split along newlines, and any concepts are mapped to code, using their to_js() function. The resulting code is then evaluated.

If this all works and raises no exceptions, a new entry is added to the history timeline below the panels. The user can review the content of the panels in the past by hovering them, and restore the panels to that state by clicking the entry. The prototype used the LocalStorage mechanism to persist the state of the panels across page reloads.

The panels present concept instances as DOM elements. When the user hovers them, a border is shown, and clicking them brings up a (non-functional) REPL to interact with the concept instance. This feature is not shown.

4.2 Interactive Mode Prototype

The second prototype has been used to explore the REPL concept of Interactive Mode. It was made after the completion of the integrated prototype and as not been included in it.

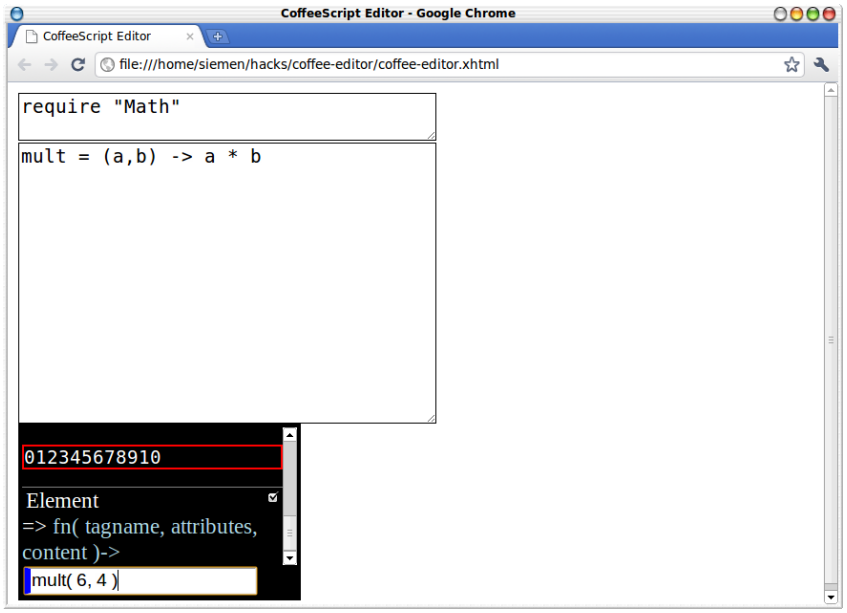


Figure 7: Prototype of Interactive Mode showing setup code, actual code, and REPL. The REPL displays replay markers, a DOM element and a function.

The prototype displays a textarea element which represents the code editor. When Ctrl+ Enter is pressed, a setup code textarea above it is shown, and the REPL below it. The contents of both setup code and main code areas is evaluated, and values declared in them can be interacted with using the REPL. Keeping statements and replaying them works as described above.

4.3 TECHNICAL FINDINGS

The implementations have shed light on the technical options for implementing a system like the proposed one in a modern browser. The main concerns turned out to be how to display Concept Instances in the DOM and performance. Other than that, there have been some difficulties with software engineering inside the browser, but these were not specific to the design of SketchCode.

4.3.1 *Displaying Concept Instances*

Newer browsers provide a built-in rich-text editor and allow making regular site content editable if the contenteditable attribute is enabled. The attribute can be overridden in containing elements. This allows

some sections to be editable, while others are not, and was used to make only the parameters of the function and variable declaration concepts editable in the integrated prototype. In this way plain text code and rich representations could be mixed. The tested browser – Google Chrome – also handled keyboard navigating across these fields gracefully, and simply skipped the non-editable parts. This also worked for concept editors that where larger than the current line and displayed as inline blocks.

Per default, the browser wraps new lines in block elements. This implies that content above the current line is seldomly rearranged when the line itself is changed, and I noted no significant layout reflow penalty.

Accessing the contents of an editor is a matter of traversing the block elements that contain its lines, and extracting either text nodes or the view DOM elements of concept instances. The latter must hold references to their concept models. The prototype used concept editors directly to hold all data for a concept. But since the DOM cannot contain a node in two places, this means that the concept can only be displayed in one place.

4.3.2 *Cursor Position*

For the autocompletion menu, I needed the current screen position of the cursor. Surprisingly, the DOM provides no way to access the current screen coordinates of the text cursor. It is however possible insert a DOM element at its current position, extract the position of this element, and remove it again. This unfortunately leads to reflows, and in effect makes the implemented prototype prohibitively slow.

This issue can be worked around either by reducing the number of time the cursor position is calculated, or by avoiding it altogether and showing the menu in a fixed position. Because the cursor's position in the DOM tree can be accessed without side effects, it is also possible to calculate it from the dimensions of the text and concept instances in the editing component.

4.3.3 *Syntax Highlighting*

Another nontrivial issue is syntax highlighting of contained plain text as it was typed. To highlight elements requires wrapping them in span elements. But since the DOM tracks cursor position relative to DOM nodes, manipulating the DOM at the cursor position loses the focus of

the editor. The informants eventually came up with a relatively elegant solution: Placing an identical element in front of the editor, with font alpha values set to zero. In this way the foremost editor catches events and holds the real content, while the backmost displays content.

4.3.4 *DOM UI Elements*

I used DOM elements as UI elements in a traditional MVC style, and e.g. styled them directly with inline attributes, instead of a CSS stylesheet. This was surprisingly unproblematic and resulted in a more object oriented way of programming than the classical page-oriented way, which tends to result in tight coupled DOM structures and CSS styles.

4.3.5 *Software Engineering Difficulties*

Other than that, implementation has not been easy, but this was not due to specific problems with the design of SketchCode, but to 'regular' software engineering problems when working in the browser.

First, both the DOM and JavaScript have inconsistencies in their design which leads to unforeseen behavior. Mistakes such as assigning a value to a system-provided attribute do not result in errors, but are just ignored.

Second, the DOM APIs are lengthy and JavasScript syntax results in code that quickly grows to an incomprehensible size. This made it difficult to keep an overview over objects, function signatures and contracts.

For the REPL prototype, I used CoffeeScript[1], a language that semantically is equivalent and compiles 1:1 to JavaScript, but provides a more convenient syntax. This noticeably moved the point at which the code became incomprehensible.

Interestingly, most of these problems could be addressed by SketchCode. This is discussed in the Evaluation chapter.

Conclusions

There seem not to be any difficulties apart from pure software engineering issues in implementing the designed prototype as a production system in a browser.

1 http://jashkenas.github.com/coffee-script/

5

THEORIES

This chapter presents a number of theories that have been used to inform the design of SketchCode scientifically. Each theory is briefly presented, and its implications on programming as a process of reflective practice is illustrated. It is shown how the theory has shaped specific features of SketchCode, and it is discussed how this differs from using a current tool.

The section is somewhat lengthy, in part because the holistic design process I argue for encourages the use of many complementing approaches. Unfortunately, qualitatively arguing for every single one of them in a report is unwieldy.

5.1 REFLECTIVE PRACTICE

The framework of Reflective Practice has already been discussed earlier. With regards to hacking, its most central aspect is the see-move-see cycle of situated learning as described by Schön [1983]. It describes a way of reaching a goal trough a cycle attempting solutions and reflecting on their outcomes to learn more about the problem and possible solutions. Whenever a move has been made, the practitioner evaluates its outcome. This is denoted to receive backtalk. Unlike feedback, backtalk requires interpretationSchön [1983]. For this, the practitioner uses his appreciative system – his ability to judge situations based on his current understanding of the problem, his experience, best practices and personal taste.

The cyclic process is necessary due to the complexity of the situation, where it is not possible to complete a task based on a complete plan.

Indications of programming being of this nature may seem obvious once one starts looking for them, but are nevertheless rarely articulated. Graham [2004] states that he "tended to just spew out code that was hopelessly broken, and gradually beat it into shape. ", and one informant noted in an interview that "debugging [was] what coding was all about".

The cycle of reflective practice is present with all tools which allow the programmer to run his program and evaluate its outcome. Interestingly, it is also absent when a program is so broken that it will

not compile and run, which deprives the programmer of backtalk and forces him to resort to mental reasoning until the program works again.

For SketchCode, the see-move-see cycle has been a guiding principle and is supported by allowing all features to provide fast and direct feedback from the machine to the programmer. An example of this is Interactive Mode and its ability to run code in isolation, which provides backtalk also from code in broken programs.

Limits to backtalk in traditional tools are caused by slow compile-run cycles, but even more often because most languages start executing from one specific entry point. If the code under question requires the programmer to put the application in a certain state before it can run, it is often more tempting to reason about the code than to run it.

5.2 REINTERPRETATIONS

In some situations, the cycle is a simple adjustment process, e.g. the adjustment of a CSS color code. But often, moves have unanticipated side effects on other aspects of the design - surprises happen. This leads the designer to reinterpret the situation and the underlying design problem. Schön [1983] denotes this seeing as. Reinterpretation and naming problems and solution attempts is one of the key aspects of the process [Cross, 2007], as it is trough this process that complete problem understanding is incrementally built. Reinterpretation is also important because it provides concepts that encapsulate parts of the design in ways that are easier to think about, and finally, because the reinterpretations trigger associations that inspire subsequent solution attempts.

Conceptual reinterpretations happen as an integrated part of the refactoring process in programming. As a case in point, the informants reported one situation where they brought an event listener/emitter framework into a shape that was so simple and general that they could reduce the complexity of their entire application considerably by arranging it according to the framework.

In the design of SketchCode, concepts are intended to support reinterpretations and display them in a way suitable to them, and not at the code level making up the concept. Concepts can be revised like any code to allow the concepts themselves to be evolved. It is however not entirely clear to me to what degree the concepts feature can accommodate all kinds of reinterpretations that occur when a program is developed.

Current systems, at least the text-based ones, make it difficult to work with reinterpretations at arbitrary levels because there are no constructs to display them. This means that the programmer must deduce them from lower level representations.

5.3 MENTAL MODELS

When humans reason about systems, they represent them internally with mental models, which they execute in mind[Schön, 1983]. The better the mental model represents the (external behaviour) of the system, the better expectations of the system the human can have. Users learn and verify their mental model of a system by interacting with it, and observing its feedback. To do this, they have to bridge the gulfs of execution and evaluation, respectively. Systems that do not allow to access some input, or that do not communicate system state after an interaction, make it hard for users to verify their mental models.

It has been shown that experience provides software designers with richer and more reliable mental models of their design material. This allows them to consider more alternatives, and to provide better and more detailed designs[Adelson and Soloway, 1985]. Thus good mental models to guide design moves are a central element of creating good designs.

On the other hand, if models are poor, the design is not very likely to function well in reality. In the case of software, mental models of components are rather complex compared to 'usual' mental models in design[Dubochet, 2009]. Agile methods [Beck, 1999]acknowledge this by forcing designs to be implemented (and thus tested) in running code. In this way the mental models are frequently checked against reality, and designs cannot for too long rely on faulty models.

SketchCode takes a similar approach in that it tries to move coding away from reasoning, towards reflective practice, to better synchronize mental models. It tries to bridge the gulf of execution trough Interactive Mode, and the gulf of evaluation trough better representing concepts using rich styled HTML views.

5.4 AFFORDANCES & DIRECT MANIPULATION

Affordances denote ways in which users perceive that they can perform actions on a system. There exist a variety of them, among others physical (e.g. pushing a door knob) and cognitive (pushing a virtual

button on a computer screen)[Norman, 2002]. The concept of affordances is related to the principle of direct manipulation [Shneiderman and Plaisant, 2004], where actions on an element on the screen are available directly on it and the user may develop a mental model of it by interacting and reversing his actions until the model is built.

In programming, the components of a program have cognitive affordances. A function affords execution or looking up its API documentation, and a variable affords logging it to inspect its value. Yet most programming environments do not provide the means to execute these cognitive affordances. They cannot run code in-place to test if the imagined behaviour is real. Looking up API documentation on a specific concept is also often happens in separate manuals, and not as a direct property lookup of the item under concern. This makes it difficult for the programmer to verify his mental model of the components of the system by interacting with it through its affordances.

SketchCode addresses this trough Concepts and Interactive Mode. Concept editors allow direct manipulation with the keyboard or mouse, and respond in ways appropriate to the current Concept. Interacting with plain text code in the REPL is a little less direct, but currently probably the best or most efficient way of interacting with a piece of code directly. Including API documentation as an attribute of functions and objects is also a way of providing a conceptual property (the documentation) trough an object's affordances.

5.5 SKETCHING

Common to design professions is the practice of sketching as a tool to quickly explore the solution space and receive backtalk. Sketching as an activity differs from the common low-fi and paper prototyping practiced by HCI in that it is more exploring and evocative. While a prototype tends towards being an experiment to test a specific assumption or design, a sketch tends more towards provoking or inspiring a subsequent sketch [Buxton, 2007]. Cross [2007]sees sketching as an intelligence amplifier, in analogy with writing. While writing support left-brain activities such as structure and logic, sketching supports right-brain cognition such as visual-spatial perception and creativity.

5.6 MATERIAL QUALITIES

Sketches are produced from a variety of materials, which are selected based on their ability to represent the aspect under question, without

requiring too much time or effort for making them. Buxton pursues the agenda of developing sketching techniques to design the user experience of digital products before they are produced. Materials that can represent user experiences, eg interactions, are eg video, role plays, or walking trough a flow of paper user interfaces.

It may be argued that sketching is a way to 'dry run' the mental models the designer has in mind. This is in fact Buxton's agenda - designingn user experiences without creating the system. The agile approach starts from the other end of the same spectrum - producing working technology, but in a way that allows it to be changed and designed into shape across iterations. With regards to programming, the limitations of the 'dry run' approach is that that there are limits to how far humans can execute mental models and receive credible feedback, while sketching in real code is more credible, but very slow.

SketchCode tries to be a material that is suitable to sketch the properties of a program. As such, it must be quick and easy to perform moves, and receive backtalk. It must also be flexible enough reshape problems, and include what is relevant for the moment, and hide what is not. Finally, it must be able to produce credible backtalk on what will work.

I have approached this problem from the agile side, grounded in running code, because program mental models are so complex, and most important properties of a program seem to emerge at run time. There exist many 'dry run' tools (flow charts, UML diagrams, static analysis), but none of them seem so well-suited as a design material as e.g. an architect's 2D renderings to judge geometrical configurations.

SketchCode supports quick moves trough Interactive Mode. Like traditional sketches it allows to some degree to leave out details, because it does not require the entire program to be runnable to interact with local changes. It supports reinterpretation through concepts. Concepts may be used in Interactive Mode, so to some degree quick sketches are not bound to only happen at the (low) text code level.

Because SketchCode takes a 'running-code' approach, it is a fairly credible sketching medium, but as a computer is always clumsy to feed data (pen an pencil are much quicker to draw with then even the best art pad) it is also a slow medium. Programmers embracing reflective practice as their process should be conscious of the sketching qualities and limitations of various materials, and supplement SketchCode with offline media (drawing charts etc) whenever suitable.

5.7 EXPLORATIVE & CONVERGING PHASES

Sketches contain only details for the aspects under question and are thus quick to make. This enables the practice of experienced designers of exploring a problem holistically and considering many alternatives[Cross, 2007]. This results in a process of exploration, but also controlled convergence, when one aspect has been refined from several angles and is found to be good enough to move on[Buxton, 2007].

SketchCode supports this process trough its History feature - a simple, yet hopefully effective, bookkeeping aid to make comparing approaches easier.

5.8 DISTRIBUTED COGNITION

Distributed cognition is a theoretical framework to describe thought processes that span several individuals and artifacts. Hutchins [1995] originally deduced it from investigating how the navigation crews of ships organized decision processes among themselves, and included constant readings of instruments and lookup tables. But it is equally applicable on a smaller scale.

Seeing coding – and the sketching of coding – as a distributed cognitive system implies that some thought actions are performed by the programmer, others by the computer. Playing to the strengths of each, the computer can take care of remembering and symbolic reasoning, while the programmer makes decision and creative tasks such as reinterpreting and coming up with alternatives. On a small scale, this already happens with modern IDEs and their context-sensitive input assistance and error checking.

SketchCode tries to split the cognitive process along this line trough Concepts. Concepts define the intent of a situation precisely for the computer and, if a reasonable editor is defined, allow it to reason about it. Conversely, the user works with a representation relevant for his part of the work - e.g. picking a color by seeing it, as opposed to imagine it and translate it to a hex code.

The drawback is of course that this scheme only works when re-using already defined concepts.

Relevance theory [Sperber and Wilson, 1986] is rooted in cognitive psychology and linguistics, and provides a qualitative model of how humans extract information from their environment. It is useful to design the extraction part; the 'see' in the see-move-see cycle, and the extraction in distributed cognition.

Relevance theory operates with two modes of converting information from the senses into useful knowledge. The first is decoding, e.g. reading, where a message encoded in (visual or audible) symbols is 'de-serialized' according to a known scheme, whereafter the receiver knows its meaning. The second mode is inference. It happens whenever the receiver combines stimuli with knowledge he already has. This results in a multiplication effect, and is usually much faster.

Relevance theory sees the human cognitive system as a bottleneck, unable to process all stimuli received by the senses. As such, it seeks to focus its efforts on those stimuli that are most likely to result in most useful knowledge per processing effort - the most relevant parts. This is naturally exploited by senders of information, who tend to indicate what pieces of sent information are most relevant, eg through body language or typography. These hints are denoted ostensions, and Sperber & Wilson see onstensive-inferral communication as the most efficient one.

Relevance Theory can explain why some programming languages and styles express a program clearer than others, and Isak Harström and I think we have provided a credible account for this this[Baader and Härström, 2009].

We analyzed how code samples that were perceived as either good, communicating, or bad, obscure, would be read according to Relevance Theory. It turned out that shorter code, which referred more to known concepts, was best because it required least decoding, and gave rise to most multiplication effects. Conversely, code that was either lengthy (required too much decoding), or used programming tricks (gave misguiding ostensions) was least efficient to read.

As a consequence of this, languages like Ruby and Scala, which have a flexible syntax and provide syntactic sugar for many concepts, and allow better ostensions by allowing special characters in identifiers, score better than languages like Java, which enforce boilerplate code to be decoded, and provide less hints in identifiers.

The fundamental observation relevant for SketchCode is here, that it must be able to accomodate reinterpretations in the form of concepts at any level - not just simple concepts like classes and language

constructs, but anything that will emerge from the design process. Concepts must be able to present only what is relevant to the programmer, and freely be able to hint relevant information. Switching to a better syntax is an improvement, but it does not scale with the emergence of new concepts. This is why SketchCode represents concepts using HTML UI elements.

5.10 CONCLUSION

Many of the presented theories encourage lines of thought that dissonate with the way contemporary programming tools work. The simplest examples of this are reenforcing mental models trough direct manipulation and affordances, but also Relevance Theory seen in the context of abstracting away irrelevant details. The theories are relatively general, cognitive theories. One one hand, this makes it likely that they apply to programmers 'in general', despite specific training or preferences. On the other hand, the risk of applying a theory in an not appropriate context is also more likely. It is therefore important to note that the theories only are used as design guides. The design itself is evaluated in the Evaluation chapter, which should provide feedback on both the concrete design, but also reflect back on the choices of theories.

6

RELATED TOOLS

A number of existing tools have either provided inspiration for SketchCode, or are relevant to keep an eye on for further work. In the following, I will present them and briefly discuss how they relate to this project.

6.1 LISP AND EMACS

The Lisp language, and its canonical development environment, Emacs, provide ways of working that are both interactive and self-extensible. Emacs allows code of the current Lisp expression to be run directly, and interacted with using a REPL. Interacting with parts of the program thus becomes an integrated part of developing.

Lisp is self-extensible because it represents both code and data in the form of S-expressions. This means that a program can read and transform a part of itself, before it is evaluated – this is a macro. Graham [1993]describes how metaprogramming using macros can transform Lisp into a language more suitable for the current task. He denotes it bottom-up programming, because the language grows up towards the problem.

The disadvantages of Lisp macros are that the resulting code is harder to debug, and that Lisp syntax in general is not very readable to most people. Because the process is very similar in SketchCode, similar problems arise. It tries to address the first issue by allowing concept editors to check validity of their data (reducing the need for debugging), and the second by allowing style HTML representations.

6.2 THE LIVELY KERNEL

The Lively Kernel [1] is an integrated system similar to Smalltalk, implemented in JavaScript and running inside a web browser. It comes with its own IDE, uses a port of Smalltalk's Morphic UI framework, mimics

1 http://www.lively-kernel.org/

Smalltalk's dynamism and conception of a running image, and can clone itself via WebDAV[Ingalls et al., 2008].

At the beginning of this project I experimented with it because the system had the potential of providing a more friendly browser programming environment than the DOM. But ufortunately it was too unstable at that point. At the same time, its rejection of the standard way of making UIs in the browser using HTML and CSS made me doubt that it would easily integrate with the existing practices of my intended users.

The project has announced that it will support standard HTML and CSS UI elements very soon, and also focus more on stability and documentation with its 2.0 release on August 1st 2011[2]. This makes it important to keep an eye on as a potential platform for further work.

6.3 INTENTIONAL PROGRAMMING

The concept of editing a program not only in code, but also rich, WYSIWYG-like representations is not new. It has been introduced by Simonyi as a way to store the intent of code directly in it - hence the name intentional programming[Czarnecki and Eisenecker, 2000].

Fowler [2005]names this category, in which he fits other tools such as JetBrain's Meta Programming System[3], Language Workbenches. Common for these tools is that they do not operate on a text based program, but on its abstract data model, which shown by the editor.

Technically, this is very similar to the way in which SketchCode stores its program's abstract structure. Here the abstract structure consists of lines of text, but where concept instances are inlined (which again may contain concept instances), the data model resembles a tree.

SketchCode uses the term Concept instead of the historically more correct Intention because the latter never became an everyday term for programmers, and more triggers association to instructions than code structures in general. Concept maps better to this. At the same time, it is not as predefined as the well-known Design Pattern, which concepts strictly speaking are instances of.

2 http://lists.hpi.uni-potsdam.de/archive/lively-kernel/2011-July/000617.html
3 http://www.jetbrains.com/mps

6.4 FIELD

Field[4] is an interactive environment for creating digital art[Downie, 2008]. It has been developed within the context of performing artists, and one key use case of it is the collaborative development of performances between dancers and programmers. In this situation, programmers need to be able to adjust their digital art and explore alternatives at the same pace as the dancers change their performance. Field provides mechanisms for this that are remarkably similar to those of SketchCode.

It offers inlining 'concepts' in the source, e.g. graph GUI elements to control wave functions, and flow charts to control the states of an installation. It integrates revision control, which is intensively used to manage branches of explored code, and to log what was evaluated. On the implementation side, Field runs off the JVM in order to integrate with existing tools, but used the Python language to quickly express code, which are pragmatic choices that resonate well with my approach.

Field itself is tailored for art installations and would not directly be the right fit for the intended use of SketchCode. But it is interesting how closely its feature set matches SketchCode's – given the fact that I first learned about it after the integrated prototype had been implemented. This may indicate that the theories of reflective practice indeed describe an efficient way of explorative programming, and that practitioners who really need this will adapt their tools to the process.

6.5 CONCLUSION

As these tools show, SketchCode is not unique in its approach. Other tools have taken similar approaches, and being aware of these has helped framing the project. Direct conclusions to the desing or implementation of SketchCode cannot be made at this point.

4 http://openendedgroup.com/field

7

ADOPTION

We have seen a number of programming styles and tools which arguably are better than the current de-facto standards, but which never have been accepted by the mainstream. But bringing improved programming practices to a wide group of users is one design goal of SketchCode. In this chapter I will deal with how programmers adopt new tools. The chapter concludes by presenting design decisions in SketchCode that intend it to support programming as Reflective Practice, and yet allow piecemeal and non-disruptive adoption by mainstream users.

7.1 HOW TECHNOLOGY IS ADOPTED

The Diffusion of Technology model describes that new technology is adopted by different kinds of users in a transition from the most innovative, to the most reluctant users[Moore, 2002]. A histogram of these groups has the form of a bell curve, and the groups are denoted Innovators, Early Adopters, Early and Late Majority, and Laggards. Innovators and Early Adopters are the first types of users to reach. A segment typically waits for the group to the left of it to adopt a technology and prove its worth, before adopting it itself.

Moore [2002]has observed that many high-tech products fail to gain mainstream acceptance, despite promising initial sales that seem to grow exponentially, but then stop abruptly.

The reason for this is that the early segments adopt high technology either out of curiosity, or because they have a very well understood need for the product's service. They are thus willing to invest more than average into learning and adjusting their practices in order to gain the benefits of the product. The majority of users is less inclined towards changes, and more interested in having their current practices supported as smoothly as possible. Because of the bell curve, sales seem to grow exponentially as the the early adopter segment is saturated. But because the needs of this group differ radically from those of the Early Majority, market penetration stops abruptly at the Chasm between the visionary Early Adopters and the pragmatic Early Majority segments.

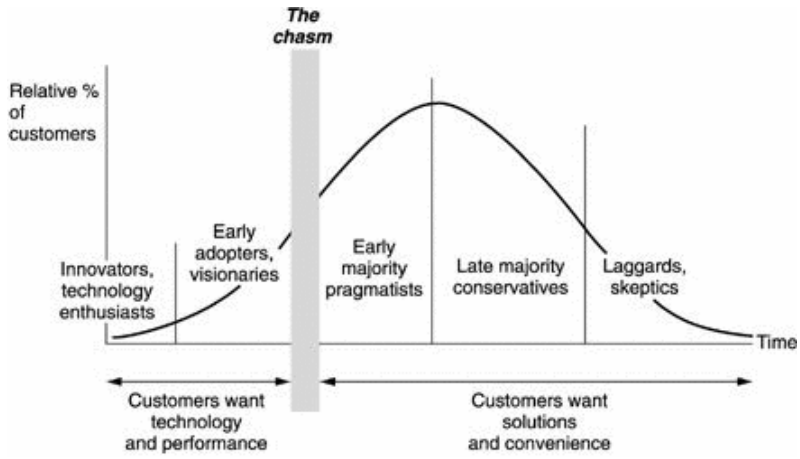


Figure 8: Adoption of technology by different segments (from http://solutions.wolterskluwer.com/blog/wp-content/uploads/crossing_the_chasm.png)

Moore's observations are specific to the high-tech industry, and Norman [1999]argues that this is due to the fact that high-tech, unlike almost any other product, is developed extremely centered around features and technical qualities. Innovators, he argues, are interested in these, while early adopters can translate these technical qualities to practical use for themselves. But the majority of users is merely interested in products supporting their existing practices without too much interruption.

This poses a dilemma for innovative technology – including SketchCode – because its very nature seeks to improve practice by changing it.

Programmers may be seen as more keen to adopt new technologies than most people. However, this does not imply that they readily will adopt any new programming tool. In fact, an informal overview[1] of the field seems to confirm that the chasm is very present here, locking the arguably better-but-odd languages into the 20% market share of the two early segments.

To deal with this problem, I present qualitative accounts for what makes adopting new programming tools difficult, and what has helped adoption.

1 http://langpop.com/

Based on personal experience and encounters with different types of programmers, I have noted a particular mechanism that works against adoption of new programming tools.

First, programming tools today are more complex than most other tools, and provide little built-in guidance for new users. This incurs more learning effort and often literally reduces productivity to zero. This differs from e.g switching from one word processor to another, and in effect actually gives programmers less incentive to switch tools than end-users.

At the same time, it is very difficult for programmers to anticipate what practical benefits the features of a new language provide. Graham [2004] has illustrated how languages vary greatly in power, because the better languages allow use of constructs that users of the less powerful languages have to implement manually. Yet, only programmers who have used a more powerful language do fully understand what use cases these features allow, and will crave them when using a less powerful language. On the contrary, a programmer unfamiliar with the powerful language looking at it will recognize the subset of functionality known from his language, along with other features perceived as incomprehensible gibberish. This is problematic even for programmers who are willing to switch, because they will not be able to know which other language to turn to.

Third, programmers can often remedy their situations with the tools they are familiar with. Unlike most end-users, they can build automation tools, plugins and more powerful libraries to assist them, while staying in the comfort zone of their known tools. This incremental improvement is perceived much more concretely than the vague possibility of maybe not even having to deal with the problem using a different toolstack. Graham exemplifies this with Lisp's S-expressions, which he argues have been reinvented, accompanied by great hype, complicated tools and a whole industry around it, in the form of XML.

Together, these three forces create a landscape of "dispersed coding islands in heavy seas". In the short term, each programmer is best off getting the best out of his current tools. At the same time, all efforts of improving by switching are difficult to plan and require great efforts to make.

At the same time, I think that we can see examples of new, improved technology that finds it way towards wide adoption despite having to interrupt business-as-usual for its users. Examples of this are to be found within the communities of the Ruby and Scala languages, and, at the risk of over-doing a frequently used example, Apple's consumer products which frequently revise established interface paradigms.

Factors that seem to support adoption are non-disruptive integration, focus on user experience instead of features and demonstrability, and self-teaching qualities.

Non-disruptive Integration

Ruby bears in it many if the consistent object orientation, dynamism and reflection qualities of Smalltalk, but is based on files files and provides efficient shell scripting abilities. In this way it integrates well with the existing operating system of its users, as well as text based RCS'es. On the other hand, the standard way of working with most Smalltalk systems is to run all code inside a virtual machine, including an IDE, but hardly interacting with the rest of the user's system.

Scala follows a similar strategy, in that it brings a more expressive language to the Java world. It runs on top of the Java Virtual Machine, is bytecode compatible with compiled Java code, and even supports Java-like coding styles. This allows Java programmers to adopt Scala feature-by-feature, integrating it with their existing projects, and staying in the same ecosystem.

User Experience Instead of Features

I mentioned that programmers who try to explore new languages have difficulties translating its unknown features to beneficial use. This is a central point in the recent 'design movement' in IT. It advocates a shift from designing features towards working with user experiences, i.e. flows from problems via features to solutions [Buxton, 2007].

When it was first released, the Rails website contained screencasts of how to implement small example applications using the framework. By documenting a user experience, it illustrated the benefit of its at the time rather odd-and-unknown DSLs and idiosyncratic constructs.

This alone is rather a case-in-point than evidence, but it seems to illustrate that a better user experience is a stronger catalyst for adoption than well-known but inefficient features. At the same time, it solves the problem that users themselves have to interpret technology and deduce their benefits.

Self Teaching Qualities

It may be safe to argue that consulting a manual before using a product is always a distraction from the task itself. Innovators might like to do it out of interest, and Early Adopters willing to put up with it. But users in the majority segments may rather give up than looking things up. Looking at programming tools from this perspective might indicate that there is a need to look into how they can become better at educating the ne user.

7.4 IMPLICATIONS

In the design of SkechCode, I have tried to take these qualities into account. The fact that it at its simplest is a plain text editor allows users adopt it gradually – they can import their old code, and do only need to introduce concepts as they emerge organically. It relies on the user's known HTML and CSS UI techniques for the same reasons. In this way it tries to mimic the non-disruptive shift of Scala. The focus on user experience, rather than features, lies in the process of Reflective Practice as a guiding theme. Directly proposing a different style of working might provoke even fiercer reactions that unfamiliar features, but also the elements of Reflective Practice can be introduced piecemeal, and conservative users can start with multiline strings and an in-place REPL. The self-teachin aspects come into play trough the focus of direct manipulation and affordances. E.g. concepts could include a built-in manual and metadata of how they can interact.

8

EVALUATION

To test whether the design is able to deliver the intended benefits, I collected real-world coding situations from the work of the informants and my own development of the prototypes. I then contrasted how these samples are worked with using current tools, and how the proposed design could improve the situations according to the presented theories.

8.1 SPECIFIC CONTEXT-AWARE ASSISTANCE

One of the most straightforward benefits of SketchCode comes from its self-extensibility. Web programming entails the use of many domain specific languages, e.g. HTML or CSS, which in Javascript are stored in plain strings. Many editors do not recognize these DSLs and treat them as normal strings. The informants made heavy use of the jQuery library, which uses XPath selectors to traverse the DOM, and complained that their editors did not support this "language in the language" with syntax highlighting. In my own coding, I especially noted how difficult it was to work with snippets of XHTML from within Javascript code, because of the lack of highlighting and closing tag matching.

A more sophisticated example of the same kind was a problem discovered when I conducted the first coding observations. One of the informants refactored nested functions. In Javascript, it is customary to use lexical scoping to bind values before an inner function is defined, and reference it in the body of the inner function. It frequently happened to him that the name of the outer variable clashed with the name of an argument to the inner function, and silently was shadowed.

Both situations could be remedied in SketchCode by creating specific concepts. Supporting highlighting of DSLs would only require using a view made from a Textarea DOM element, and implementing a simple keyword highlighter. The informants did already often tinker with their tools, and were well capable of this[1].

1 One informant e.g. extended his TextMate editor with a feature to evaluate selected JavaScript code.

Detecting variable shadowing would be a little more complicated, but in essence work in the same way. It would require defining concepts for functions. Variables and arguments with clashing names could then be detected by parsing the source code in the function concept editor, e.g. using a tool like the Narcissus JavaScript parser, and indicating conflicts visually. An alternative would be to define concepts for variable declaration, and use them within the body of a function. In this way, the function concept instance could directly access the names of contained declarations trough its contained declaration concept instances.

In both cases, the users would be able to incrementally improve their editors by using the web UI technologies that they are familiar with. If it is easier, it may be done more often, and might in effect result in users having better tools for their particular situations. Perceiving one's tools as something malleable might also lead to users reflecting more on the structure of their programs, asGraham [1993]has argued.

But seen from the perspective of the presented design theories, the potential of this kind of use is limited. It resembles the already happening incremental improvement of editors by users themselves, and does not move programming radically towards reflective practice.

8.2 HIGHER LEVEL CONCEPTS

A more potent way of using custom made concepts is to use them to hold higher level structures of the program, and represent them in more comprehensible ways.

8.2.1 *Visualizing Control Flow*

JavaScript handles concurrency trough events and callback functions. This leads to a continuation passing style (CPS) of programming, where long-running functions return immediately, but take a function as an argument that is invoked when the long-running operation is completed. The informants relied heavily on CPS programming, but complained that it easily leads to code that "drifts to the right" of the screen, because it leads to many nested callback functions, which are hard to read and introduce the mentioned risk of variable shadowing.

Several utilities deal with this, either by parsing and re-writing code to transform it from a sequential-looking structure to CPS form (Streamline.js[2]), or by letting the user declare all callback functions in an array,

2 https://github.com/Sage/streamlinejs/blob/master/README.md

and then rendering it to CPS form (The Common.js[3] step function). In fact, the informants started using the latter.

The basic problem in this situation is that the sequential control flow of the program branches, which is conceptually simple, but causes syntactic noise. Using SketchCode, the problem might have been addressed by defining concepts for functions that perform asynchronous calls and thus branch the control flow. The two branches would then be contained in textboxes, shown side by side or rearranged as desired.

8.2.2 *State Charts*

One situation I encountered very often when implementing UI based prototypes was the difficulty of dealing with UI states and the flow between them. The dropdown autocompletion menu was one situation that depended on several states, including the current word under the cursor, the current selected item, and if the user had closed the menu by pressing Escape. Another was the REPL feature, that allowed the user to move back in the entry history by pressing arrow keys.

Common for these situations was that the user's flow trough them could be gradually worked out with the help of a state chart on paper, which I then implemented in code. This process was tedious and error-prone. Often the desired behavior of the UI element only became clear by interacting with the running program, which meant updating the chart and propagating the changes to code. In addition, coming back to the code after some time made it very difficult to comprehend the structure and map it back to a state diagram to reason out additions and their consequences on the existing code.

In this particular situation, SketchCode would have allowed me to define a concept to define abstract state machines. Such a concept would hold a model representing the states, transitions and events that trigger them. The model would be shown graphically, e.g. as a SVG diagram, and allow the creation of additional states, transitions and events at the chart level. Code to be executed on transitions and to test for events would be entered into text fields on the chart. At runtime, the concept would render to a JavaScript function containing one large switch statement that handles all defined cases.

This would have freed me from the process of transforming the state chart into code, and left me with interacting with the running program to figure out the exact desired behavior of the UI element, updating the parameters of the state machine incrementally.

3 http://www.commonjs.org/

Common for these two examples of higher level concepts is that they represent problems that exist often in coding – overlooking concurrency and state permutations –, but which most programming environments do not address directly. In a sense, programmers deal with high-level concepts, but have to elevate them from the code-centered representations of most tools in their heads. Like (one aspect of) sketching, defining custom views allows them to deal with the relevant aspects of the problems represented in a way that helps them. The transformation to computer-readable representation is uniform but tedious, and automated. Interpreted as a situation of Distributed Cognition, this division of labor between man and machine plays to respective strengths.

8.3 INTERACTIVITY

Coding UIs in JavaScript often entails writing callback functions that close over values defined outside the function, as well as the event parameter. I encountered several such situations while coding the autocompletion menu. A typical way of working with this for me was to implement the callback gradually, and placing log statements in it to ensure that the call happens as intended. However, this is problematic when the callback itself contains code that needs to be developed gradually – e.g. the slicing, mapping and accumulation of a list (to mach concept names, in this instance). This results in a coding situation where the program needs to be built 'from two directions'. The typical fix is it to copy the code to a place where it can be referenced, and copy it into the callback when it is ready. This is tedious, and becomes especially annoying when debugging, because the code then has to be moved out of its place to be interacted with.

The Interactive Mode would have been helpful in this situation, because it would be able to run the program pieces from both directions, without having to move them out of place.

8.4 EMPOWERMENT

It was very important for the informants to work with simple, fast and generic tools. One informant stated that he ideally only used the features of a code editor that he also had available over an SSH terminal link. Another reported that he had given up on Emacs because he always had to consult the manual when he wanted to use one of its (admittedly powerful) features, and Eclipse because it was so slow and confronted him with all kinds of dialogs he did not want to bother

with. They preferred dynamic languages that could be run directly, and interacted with in a REPL, and were initially skeptical of SketchCode because it had graphical elements. They had used graphical programming languages, and often found them cumbersome to use because these tools let them only wire together predefined concepts in a tedious point-an-click fashion, not fast typing on a keyboard.

These issues may be seen as common usability issues. Systems that respond slowly, cannot readily be interacted with, or confront the user with undesired information, frustrate their users because they cannot readily be controlled [Norman, 2002]. Many of these issues must be taken care of during implementation, but I will briefly discuss how they are dealt with in the design.

8.4.1 *Direct Keyboard Interaction and Speed*

Concept definitions are inserted using the autocompletion dropdown menu, because this feature already is used efficiently in text editors. It allows the user to summon concept instances without taking his hands off the keyboard. Simple concepts that just define input fields within non-editable areas, like those in the working prototypes, can be navigated using the keyboard arrow and tab keys. Ultimately, it is up to the implementer of concepts to deal with the input mode of each individual concept. This allows the programmer both to break, but also to optimize quick keyboard navigation of concept instance editors.

Graphical concepts like e.g. the state chart are more difficult to support trough keyboard input, but there is no reason to expect that mouse input per-se would be less efficient for this kind of concepts.

In any case, it is important that navigating the SketchCode interface works usability-wise well, and that e.g. the focus does not get "stuck" in a particular concept editor. The implemented prototype used a mixture of editable fields, plain text, and non-editable concept representation inside a **contenteditable** block element, and navigating these with the kayboard went reasonably well out-of-the box.

Likewise, the technical prototypes showed that, even though the implemented prototype was slow, it would be possible to implement a reasonably fast editor. The browser as a platform has (speed) limits and will probably never reach the speed of a local editor implemented in C. But web applications like GMail and Google Docs provide feasible user experiences, and the reflow behaviour observed in the prototype does not indicate that the same speed cannot be reached because of the type of content that SketchCode shows.

Thus, the fact that SketchCode uses rich representations does not have to imply poorer UI interaction.

8.5 CONCLUSION

Confronting a design with data from real life situations is probably the most reliable way of testing it. As presented here, it seems indeed like there are credible, empirical use cases for the intended design, and it would definitely be worth implementing a functional prototype to explore these scenarios further. On the other hand, one must keep in mind that these individual situations only confirm usefulness in the specific cases; they do not validate the support of the entire process of programming as reflective practice.

9

DISCUSSION

In the previous chapters, I have addressed the making of a new kind of programming environment based on the cognitive styles of designing from several angles – epistemology and method, technology, theories of use, related work, and probabilities of adoption, and finally evaluation. I will now discuss the bigger picture that emerges in summing up these findings, as well as reflections on the process itself.

First, the overall picture of the SketchCode approach seems promising. From the conducted evaluation, and also from that fact that similar approaches (e.g. Field) seem to work for their users, it seems likely that there is value in pursuing this path further. The current barely-functional prototype, and the illustrations, have reached the limit of how much feedback they can provide on the design proposal. A more credible inquiry now requires the implementation of a usable prototype, which users can interact with.

The discussion on adoption was likewise promising, but based on very little empirical data. A working prototype could be used to observe how disruptive the proposed design would be on user's practices. This would allow a better qualitative assessment of adoption possibilities, because the argued non-disruptive adoption strategies would be empirically grounded. Such an assessment would still be based on the technology lifecycle model.

Regarding the process, the use of a design process has both been rewarding and challenging. Rewarding, because it indeed has brought the design, and my understanding of how coding and reflective practice go together, forward in a way that analysis would not have done.

Challenging, in that the intended holistic approach has led to many slightly-too-large subprojects. especially, it seems problematic that I intended to liv up to academic transparency standard, because much desing data and also decisions are hard to articulate, and hard to present credibly in the form of a report.

10

DIRECTIONS FOR FURTHER WORK

I suggest that a working prototype of the proposed design is implemented, to test it in practice, and to study user reactions.

BIBLIOGRAPHY

Beth Adelson and Elliot Soloway. The role of domain experience in software design. *IEEE Trans. Software Eng.*, pages 1351–1360, 1985.

Siemen Baader and Iask Härström. The communicative aspects of programming language, 2009. URL http://itu.dk/people/siba/papers/The%20Communicative%20Aspects%20of%20Programming%20Languages.pdf.

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, us ed edition, 1999. ISBN 0201616416.

Bill Buxton. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann, first edition, 2007.

Nigel Cross. *Designerly Ways of Knowing*. Birkhäuser, 2007. ISBN 978-3-7643-8484-5.

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 0-201-30977-7.

Marc Downie. Field - a new environment for making digital art. *Computers in Entertainment*, pages –1–1, 2008.

Gilles Dubochet. Computer Code as a Medium for Human Communication: Are Programming Languages Improving? In Chris Exton and Jim Buckley, editors, *Proceedings of the 21st Working Conference on the Psychology of Programmers Interest Group*, pages 174–187, Limerick, Ireland, 2009. University of Limerick. URL http://www.csis.ul.ie/PPIG09/.

Martin Fowler. Language workbenches: The killer-app for domain specific languages?, 2005. URL http://martinfowler.com/articles/languageWorkbench.html.

Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice Hall, September 1993.

Paul Graham. Succinctness is power, 2002. URL http://www.paulgraham.com/power.html.

Paul Graham. *Hackers and painters - big ideas from the computer age.* O'Reilly, 2004. ISBN 978-0-596-00662-4.

Edwin Hutchins. *Cognition in the Wild*. The MIT Press, new edition edition, September 1995. ISBN 0262581469.

Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. The lively kernel a self-supporting system on a web page. In Robert Hirschfeld and Kim Rose, editors, *Self-Sustaining Systems*, volume 5146 of *Lecture Notes in Computer Science*, pages 31–50. Springer Berlin / Heidelberg, 2008. URL http://dx.doi.org/10.1007/978-3-540-89275-5_2.

Erik Stolterman Jonas Löwgren. *thoughtful interaction design: a design perspective on information technology*. MIT Press, 2007.

Geoffrey A. Moore. *Crossing the Chasm*. Harper Paperbacks, revised edition, September 2002. ISBN 0060517123.

D. A. Norman. *The invisible computer : why good products can fail, the personal computer is so complex, and information appliances are the solution*. 1. mit press paperback ed edition, 1999. ISBN 0-262-14065-9.

Donald A. Norman. *The Design of Everyday Things*. Basic Books, New York, reprint paperback edition, 2002. ISBN 0-465-06710-7.

Donald A. Schön. *The Reflective Practitioner*. Basic Books, 1983. ISBN 0465068782.

Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001. ISBN 0130676349.

Ben Shneiderman and Catherine Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction (4th Edition)*. Pearson Addison Wesley, 2004. ISBN 0321197860.

Dan Sperber and Deidre Wilson. *Relevance*. Blackwell, 1986. ISBN 0631137564.