Author:
Tim von Oldenburg

Supervisor:
Lars Holmberg

Examiner:
Jonas Löwgren

May 26, 2014

Making scope explorable in Software Development Environments

to reduce defects and support program understanding

Thesis Project I

Abstract

Programming language tools help software developers to understand a program and to recognize possible pitfalls. Used with the right knowledge, they can be instrumented to achieve better software quality. However, creating language tools that integrate well into the development environment and workflow is challenging.

This thesis utilizes a user-centered design process to identify the needs of professional developers through in-depth interviews, address those needs through a concept, and finally implement and evaluate the concept. Taking *scope* as an exemplary source of misconceptions in programming, a „Scope Inspector“ plug-in for the Atom IDE—targeting experienced JavaScript developers in the open source community—is implemented.

Acronyms

API Application Programming Interface

APM Atom Package Manager

AST Abstract Syntax Tree

CSS Cascading Stylesheets

DOM Document Object Model

HTML Hypertext Markup Language

IDE Integrated Development Environment

JIT Just-In-Time

JVM Java Virtual Machine

OSS Open Source Software

PARC Palo Alto Research Centre

UCD User-Centered Design

UI User Interface

WYSIWYG What You See Is What You Get

Contents

1 Introduction

Creating computer programs is a difficult and complex process. Integrated Development Environments (IDEs) integrate a number of tools helping software developers do their work and are indispensable in a modern development workflow. One class of those tools are *language tools*, which help with the actual programming language itself and reduce defects and misconceptions (Hidayat 2012). However, research by Johnson et al. (2013) on static analysis tools suggests that they are not as widely used, although they proof to be helpful. This thesis attempts to approach the design of language tools by means of interaction design methodology, through a user-centered design process and by the example of *scope*.

In programming, scope is an abstract concept to define the validity of variables. By looking at the structure of scope, a program can be explored from a different perspective than just its source code and symbols, and certain pitfalls that lead to program defects can be uncovered. In languages that implement lexical scoping, such as JavaScript, scope analysis can be done using static analysis, and can therefore be applied during author-time already. The research that was done in the course of this thesis shows that scope is not yet appropriately addressed by language tools. It is therefore used as an example for designing, implementing and evaluating a language tool targeting professional developers. A more in-depth explanation of scope is given in chapter 4: *Relevant Programming Concepts*.

1.1 Process

This thesis project follows a User-Centered Design process, which is as well reflected in the structure of this document. Preliminary theoretical groundings are presented in chapter two, which introduces software development environments and their history and role in the development workflow. It also presents relevant concepts of programming languages, scope and its implications in particular. Chapter three describes the exploration phase by means of a survey and interviews with professional software developers to identify characteristics of well-integrated development tools. Furthermore, canoncial and related work examples are identified and listed, and the ideation process is exposed. The design itself is conducted in three iterations: sketches, a scripted prototype and a working prototype. Chapter four presents these iterations and explains the design decisions that have been made. Integrating a solidly implemented, high-level prototype with the Atom text editor demonstrates the feasibility of the concept, which is further verified through user testing. Findings from this phase are finally discussed in the fifth chapter.

The project described in this thesis targets the needs of professional developers with advanced experience. The final design is built for the JavaScript programming language, but the concept of scope presents difficulties in nearly every language in use. The knowledge gained during the process is thus expected to be applicable to other programming languages as well. It will be validated by means of both quantitative and qualitative data using analytics, general feedback on the web, and interviews.

1.2 Knowledge Contribution and Limitations

This thesis explores how language tools for professional developers can be designed and evaluated. It creates knowledge in the field of interaction design by contributing the following:

- Characteristics of well-integrated language tools that are important for professional developers to support their work.
- A way of evaluating designs with a specific, professional target group
- The implications of testing prototypes with a very limited user group in the open source community
- Using an interaction design approach to create open source software opens up the field. It yields results that are most probably different from what typical innovation processes in open source would have resulted in.

In addition to the interaction design knowledge, this thesis makes the following contributions to the open source community:

- A working, extendable prototype in the form of a plug-in for the Atom IDE. The plug-in is released as open source.
- A static analysis library to extract relevant JavaScript scope information. The library is written in CoffeeScript, released with the Atom plug-in[1] and can theoretically be re-used in any software to analyze scope in JavaScript.

Limitations

The short time period in which this project was pursued (8.5 weeks) creates some limitations. On the one hand, the final prototype—though being a high-fidelity working prototype—has a limited set of features as well as some bugs that influence the evaluation outcome. On the other hand, the focus of this thesis has to be very narrow, which is why some of the findings are difficult to apply to other programming environments, programming languages, and less experienced developers.

[1] It is planned to extract the library from the plug-in in the future.

2 Research Framework

This chapter introduces the research done prior to the design. It presents a short history of IDEs and list typical User Interface (UI) design patterns in IDEs.

2.1 History and Purpose of Integrated Development Environments

> „A programming environment is a user interface for understanding a program.“ — Bret Victor (2012)

Software development environments have been predecessed by general text editors, starting with several projects at the Xerox Palo Alto Research Centre (PARC). Douglas Engelbart created the text editor for the NLS system (oNLine System) which allowed editing with direct manipulation and What You See Is What You Get (WYSIWYG). In the *Gypsy* text editor, Larry Tesler first integrated modeless moving of text, which is known as *Copy & Paste* (Moggridge 2007). Text editors with those functionalities are now the core of any software development environment.

Later, while working with Alan Kay, Tesler created the first class browser for the Smalltalk programming language. Class browsers are used to look at programs not as of textual source code, but as of logical entities of a programming language (for example classes and methods). The Smalltalk class browser was therefore the first software specifically written for creating software, and a predecessor to any modern development environment.

IDEs integrate text editors (due to their specific purpose also referred to here as *code editors*) with other software development tools. Typically, those tools include compilers, build systems, syntax highlighters, autocompletion, debuggers, and symbol browsers. The first IDE is said to be *Maestro I* by Softlab, a whole terminal dedicated to integrating various development tasks (*Interaktives Programmieren als System-Schlager* 1975).

2.2 IDEs compared to Text Editors

It is difficult to delimit the term „Integrated Development Environment“ and contrast it with text editor that are mainly used for programming. Baxter-Reynolds formulates a basic definition:

> „What the different is between a text editor and an IDE – to me at least – is that an IDE understands the language, whereas the text editor understands text." (2011)

In his article, Baxter-Reynolds tries to make a point against the use of text editors for programming by stating that an IDE brings „forward an understanding of the underlying language and the structure of code, and puts it front-and-centre in your working environment." (2011) While certainly being correct with this point, he ignores situations where the „understanding of the underlying language and the structure of code" is either not wanted[1] or not possible to achieve.

According to Lynch (2011) the latter is often the case in web front-end development. Through working with lots of different file types and programming languages, neither of which dictates a certain structure (in opposition to many static languages like Java), an IDE can only have a limited understanding about the structure of the code. Lynch also states that IDEs „tend to be built with a workflow in mind", therefore being seen as opinionated.

In other words, IDEs and text editors seem to follow different, contradirectional approaches. While the latter is built around a central paradigm (text editing) and usually comes with a minimal program core that is extendable to personal likes, IDEs tend to offer everything ‚out of the box' as a one-stop solution.

For this thesis, the distinction only plays a subordinate role, as most of the concepts and ideas discussed here can be applied to both kinds of software. However, it is important to clarify that both are adressed when using, interchangably, any of the following terms: *Integrated Development Environment (IDE)*, *development environment*, *software development environment*, *programming environment*.

2.3 The Current Landscape of Development Environments

The IDE landscape is today more differentiated than ever, ranging from minimal, purpose-specific editors to full-fledged, general-purpose, commercial development environments. This diversity can also be seen in the survey results (see chapter 3).

On the commercial side, Microsoft Visual Studio is the monopolistic development environment for the .Net platform. The Java platform is dominated mostly by open source IDEs, such as Eclipse and NetBeans, although they and their derivatives are widely used for other programming languages as well. More specialized are the Processing and Arduino environments. JavaScript and web frontend developers, however, often use more minimalistic code editors, for example Vim or Sublime Text.

Those different IDEs serve the needs of different developers and development situations. But still, it seems like there are many niches that are yet to be filled with new environments. Especially the area of web development (frontend development) experiences many new products quite often, which is possibly related to JavaScript's growing importance as the language of the web. The latest additions to the row of

[1] For example, because it may collide with other features that have a higher priority for the respective developer.

web-focused IDEs include Github's Atom, Adobe's Brackets and Eclipse Orion, all of which are based on Node.js and other web technologies.

There are also recent developments in different development paradigms. DeLine et al. (2006) and Bragdon et al. (2010) introduce novel user interface metaphors to structure and navigate program code. *Code Bubbles* by Bragdon et al. provides a prototypical implementation, and a similar concept is adapted by Granger (2012). *Code Thumbnails*, as presented by DeLine et al., is implemented in Sublime Text.

2.4 Interface and Interaction Patterns in IDEs

Many User Interface patterns found in IDEs are general, well-known UI patterns adapted to a specific purpose. This section gives an overview on interaction patterns in IDEs that are relevant to this thesis.

2.4.1 User Interface Patterns

Code Editor Central to every IDE, a code editor is a specialized text editor, used for reading and writing program code. It typically features a *gutter* (see below) and Syntax Highlighting. In opposition to the text editor of a word processor, code editors are not rich text editors. They also display a monospaced font, which allows to see the editor content as a grid of rows and columns. With evenly-spaced columns, due to the monospaced font, code formatting and line indentation[1] is made consistent.

Gutter The gutter is part of the code editor and describes the narrow space next to the actual code (usually to the left). Gutters are mainly used to display line numbers (important for navigation and debugging), but some provide more advanced features, for example setting breakpoints[2], indicating errors in the code through symbols, showing version control information, or allowing to fold code away in order to either focus or get an overview.

Panel (sidebar) A panel is a rectangular UI area used to group together interface element of similar functionality or other commonalities together. Often, panels are used on the edges of application windows; if they are on the left or right side, they may be called *sidebar*. Panels that host a great number of program functionalities are often called *toolbar*. Some applications implement *dockable* panels, which can be moved around and snapped to different areas on the screen. Another common characteristic is that panels can be resized and *toggled*, i.e. shown and hidden, on demand.

[1] In many programming languages, line indentation is an important concept, either as a core syntactical concept or for the sake of readability.

[2] A feature of the debugger; when set, the program stops at the specified line to allow step-by-step investigation.

Status bar The status bar is known from many programs, for example web browsers and word processors. It is a small bar (about one text line of height) at the bottom of the program window, usually spanning the whole window width. It is mainly used to display status information and quickly switch between different application modes (for example „insert“ and „overwrite“ in word processors).

2.4.2 Interactional patterns

Navigation Usually, code can be both browsed and searched for from different perspectives.

For browsing, most IDEs have a built-in file browser. IDEs that have the respective understanding of code structure can also offer a more *logical* way of navigating, for examply by symbolic entities like modules, classes and methods. Those are usually listed in a symbol browser or class browser. In the Eclipse IDE, the file browser and symbol browser are combined into one component, called the *project explorer*.

IDE facilities for searching work analoguously. Files within a project can be searched for by their name or their content. If the IDE knows about the symbols of a programming language, those can usually be searched for as well. Additionally, some IDEs like Eclipse allow the user to right click on a method call and jump to its definition source file, if available.

Modes In most IDEs, UI elements can be shown or hidden, sometimes even positioned anywhere on the screen. The Eclipse IDE even allows the creation of completely different UI configurations, so-called *perspectives*. Usually, perspectives are build for a certain task, e.g. developing or debugging. Text editors like Sublime Text and Atom[1] support a so-called *distraction-free mode*, in which all User Interface elements are hidden except the editor itself.

Input Most modern IDEs are mouse-driven, which means that every goal—except writing code—can be achieved using the mouse alone. However, as users proceed to become more familiar and proficient with the IDE, they tend to utilize keyboard shortcuts to be more efficient. Many IDEs allow the user to configure keyboard shortcuts freely; others even offer a single shortcut to reach any menu item through fuzzy searching, for example Sublime Text.

Execution, Evaluation and Debugging Most IDEs allow the user not only to edit a program, but to compile, run, and debug it from inside the IDE. This has the advantage that any information related to compilation-time and run-time can be used and presented in the IDE itself. In its simplest form, compilation errors or the console output of a program is shown in an extra output area on the screen. More advanced implementations show debugging or compilation informatin *inline*. To give an example: if the Java compiler in Eclipse encounters an error, it lists the error in an extra

[1] In Atom, this has to be installed through a package: `https://atom.io/packages/zen`

panel, but also underlines the affected code with a red line. For more information concerning the different lifecycle phases of a program, please refer to chapter 4: *Relevant Programming Concepts*.

3 Exploration

Following the research framework, this chapter presents the exploration phase of the design process. Through an initial survey and a sequence of interviews, a whole range of problems is explored. Finally, the problem of scope is put into focus for this thesis.

To form a general understanding of how IDEs and some of their specific features are used, an online survey targeted towards professional developers was created. The survey ran in April 2014 over the course of two weeks and yielded answers from 45 participants.

Besides general questions, e.g. which programming languages and IDEs the participants used, it collected information about the usage of the following IDE functionalities:

- Navigation of code
- Debugging
- Application Programming Interface (API) and language documentation
- Autocompletion
- Project structure and scaffolding
- Asynchronicity
- Syntax Highlighting

For each of the areas the survey asked if and how the participants were using them and—if appropriate—how their IDE was supporting them. The survey instrumented multiple-choice questions with an additional „Other“ field for custom answers, as well as open-ended questions with a free-form text field.

3.1 Survey Results

The survey participants listed 21 different programming languages they are using on a regular basis, as well as experience in 19 different development environments. The participants' background is diverse, although the major part seems to be working with web technologies (both front-end and back-end).

About 76% of the participants look up API and language documentation mainly on the web, only a small percentage uses the integration of documentation into IDEs. However, nearly every participant (87%) makes use of the IDE-provided autocompletion feature, although most of them came up with ways to

improve it. Many comments are directed towards „smarter“, more context-aware autocomplete suggestions, up to levels of artificial intelligence. Some comments also mention a lack of performance and subtlety.

Navigation within large code bases is done in many different ways, such as presented before: file browsers, symbol browsers, file search or content search. However, there does not seem to be a clear general preference. For structuring code, most participants rely on platform-given modularity, for example through packages and classes in Java. In programming languages where project structures are not given, developers use frameworks and design patterns to achieve a similar structural consistency.

All participants value syntax highlighting, although for different reasons. However, some offered suggestions on how highlighting of certain code tokens could be used otherwise to reduce errors. Two suggestions were targeted towards highlighting of *similar* identifiers in order to recognize typos. Others, however, intended to focus on semantics instead of syntax; for example, indicating value changes of the *this* keyword in JavaScript, highlighting the currently focused block of code, or colour-coding the relationship of interdependent variables.

3.2 Interview Results

In succession to the survey, ten participants agreed to be interviewed, seven of which the author conducted interviews with. The interviewees are either currently working as full-time or part-time professional developers or have been doing so in the past and are now in related positions such as *IT consultant*. Aside from that, their backgrounds are diverse, ranging from part-time front-end developers with a focus on design, over web application developers to low-level audio specialists. They have experience with 12 different IDEs, using 15 programming languages on three different operating systems. Below is a summary of interview results that are related to the thesis project.

Nearly all the interviewees stressed the importance of *performance* in any software development tool, especially in IDEs and code editors. If a feature is too slow, reacts too slowly or slows the overall IDE down, it is considered disturbing to the development workflow. Especially the web developers praised lightweight code editors, favouring them over the more heavyweight IDEs, but still recognizing their drawbacks: lightweight editors are not as *smart* (see below).

Most of the interviewees also referred to their development tools of choice in regards to the *Unix philosophy*, which—according to Ken Thompson—states that programs should „do one thing and do it well.“ (Raymond 2003) This thought ultimately leads to modularily designed systems, which was stressed in the interviews in different forms. Most obvious is the ability to enable and disable features, as well as some kind of plug-in management in general. Two interviewees also mentioned *modes*, although in different contexts. On the one hand, features could run in different modes to provide help or stay out of

they developer's way (*beginner and expert modes*), on the other hand modes could be used to get a different perspective on a program (e.g. highlighting of different aspects in the code).

The interviewees expect their development environment to behave in a *smart* way; it should ideally know beforehand what the developers need in terms of support in a given situation, and what code they are about to write. On first look, and given the current landscape of IDEs and text editors, this contradicts the desire for a lightweight, fast, unopinionated development environment. To be smart, a development environment must have knowledge about the programming language, the libraries used, and about best practices. Some IDEs are tightly integrated with their target programming platforms, for example Microsoft Visual Studio with the .Net platform and Eclipse with the Java platform. But these programs are generally not considered lightweight, performant or unopinionated. *Smartness* for lightweight environments, however, can be achieved by combining it with the modular approach of the Unix philosophy. If specialized programming language tools can be loosely plugged into lightweight development environments, smartness can be achieved in those environments as well. A good example for this is given by the numerous *Linter* plug-ins for editors like Sublime Text (see section 5.1: *Canonical Examples*) and projects like CTags[1].

The last relevant result of the interviews to be mentioned in this section is a *focus on code*. No matter the target platform and the developer's background, code is still in focus of the development process nowadays, which makes the code editor the most important part of any development environment. The term *inline* describes activities that happen within the text editor itself, for example syntax highlighting. If development tools work inline, the developer does not have to switch focus back and forth from the authoring process. However, by putting additional information inline, there is a risk that the text editor becomes too cluttered or visually busy, confusing and distracting the developer. This is exemplified by pop-up windows that block a lot of editor space or additional coloured text that makes colour-coding ambiguous. Thus, programming tools that display information inline must be carefully designed to be unobtrusive.

Four important characteristics for programming environments can be extracted from the interviews: *performance*, *modularity*, *smartness* and a *focus on code*. Integrating these characteristics is important for the usability and usefulness of development environments, and thus for any tools that enhance them.

3.3 Scope as a valid problem

Through the conducted interviews and the survey, one can argue that *scope* is a promising and valid problem area to explore. Although it was not referred to in the survey in any way, *scope* was mentioned independently by several of the survey participants and interviewees in suggestions for the improvement of existing patterns and tools. One of the interviewees introduced Crockford's (2013) approach of *context*

[1] See `http://ctags.sourceforge.net/`

colouring (see below). A similar approach was suggested in the survey in the context of editing. Though not necessarily related to scope, the participant suggested to highlight the current code block the cursor is placed in; this is already done by some editors and IDEs, and is adapted in the concept presented in this thesis as well. Another interviewee suggested to indicate changes of the *this* context in JavaScript, which is closely related to scope, although being run-time specific.

The strongest alternative problem to possibly focus on was *asynchronicity* and the writing and debugging of asynchronous code. After some research on the topic, I found the work of Lieber, Brandt & Miller (2014) to be quite substantial and possibly parallel to my then-prospective work. Lieber implements *Theseus*, an asynchronous JavaScript debugger, and will discuss it from an Interaction Design perspective in his forthcoming master's thesis. This is why I did not choose the topic of asynchronicity, but instead focused on the problem of scope.

4 Relevant Programming Concepts

In the following chapter, the design framework is further narrowed down towards relevant concepts of programming languages, especially *scope*. The research done in chapter 2 regarding software development environments still remains valid and relevant; this chapter only adds a new, very narrow dimension to it.

Whereas most of the concepts presented below apply to a wide range of programming languages, *JavaScript* was chosen as an exemplary language both to explain the concepts as well as the target language of prototyping as described in chapter 6: *Design Iterations*. The reasons for this choice are my familiarity with the language, as well as the fact that JavaScript is one of the most ubiquituous languages used due to its role in the world wide web and its implementation in web browsers, respectively.

4.1 Program Lifecycle

The lifecycle of a computer program consists of different phases, the most relevant of which are described briefly in this section.

Author-time shall be the phase during which a program is written, read, understood, and edited. There is no canonical definition or common name for this class of activities around source code, which is why I define *author-time* as the time separate from run-time in which a program author (e.g. a developer) deals directly with its code. An alternative name for author-time may be *edit-time*, *creation-time* (Simpson 2014) or *construction* (McConnell 2004).

Compile-time is the phase in which program code is translated (compiled) into native machine code or an intermediate representation (e.g. Java Bytecode in the case of the Java Virtual Machine (JVM)). This process generally consists of lexical analysis, parsing and code generation.

Run-time is the phase during which a program is executed. In some interpreted languages, Just-In-Time (JIT) compilation[1] leads to a convergence of compile-time and run-time, which makes the distinction harder. *Run time errors* are errors happening during run-time that could not be detected during compilation (for example if they depend on user input).

[1] Just-in-Time compilation is the compilation of code immediately before its execution, instead of during a preliminary compilation phase.

Debugging is the process of identifying and eliminating software errors, so-called *bugs*. This activity is usually supported by a specialized software called a *debugger*. The debugger allows to hook into a program during run-time through so-called *breakpoints* and step through each statement individually. At all times, the debugger can expose the values of variables in the respective context.

This thesis and the according prototype mainly address the author-time phase, during which so-called static analysis can be performed.

4.2 Scope

Scope is the part of a program in which a given variable is accessible. In computer programming, variables are used to address (write and read) data. At some point in the program, a variable is *declared*, i.e. its existence is made known to the program. However, in most programming languages, a variable declaration in some part of the program does not necessarily make the variable accessible from *all other* parts of the program. The area in which the variable is accessible is called its *scope*.

According to Simpson (2014), scope is „the set of rules that determines where and how a variable (identifier) can be looked-up" and therefore be accessed and used. The specifics of „where and how" depend on the respective programming language. Most modern languages implement *lexical scope*, which means that the scope of a variable depends on the position of its declaration in the actual source code. In other words, where in the source text a variable is declared defines also where it is usable and accessible.[1] Lexical scope also means that scope is defined during author-time already, and can thus be analyzed early on. In contrast, the *this* keyword in JavaScript is a run-time phenomenon; its value cannot be known during author-time.

As scope is a concept that is central to a program, it can be used as a perspective to look at a program, too. The most obvious perspective is *source code*. Code is organized in different files, and files are lines that run from top to bottom. Another way to look at a program is by its symbols, for example modules, classes, methods. Java programs are organized in packages; each package has several classes, of which each has attributes and methods. Finally, programs can be looked at by means of scope, which has its own characteristics. Those are described in the following sections.

4.2.1 Nested scope & variable lookup

Scope is a hierarchical concept: in many programming languages, scope can be nested by creating a scope *within* another scope. Consequently, we will use the following definitions throughout this document:

Child scope A scope `b` created immediately within another scope `a` is a child scope to `a`.

[1] The complementing concept, *dynamic scope*, is not relevant to this thesis.

Descendant scope Any scope nested inside of a scope `a` is descendant to scope `a`.

Parent scope The scope in which an immediate child scope is created is its parent scope.

Ancestor scope If scope `b` is a descendant to scope `a`, `a` is an ancestor of scope `b`.

Scope chain Given a scope `a`, the scope chain of `a` is the set of nested scopes from `a` up to the global scope.

In JavaScript, scope nesting is an important concept for variable lookup. When the JavaScript engine encounters an identifier, it looks for this identifier in the current chain of scopes. For example, if a variable is used in a scope `a`, the JavaScript engine first looks for its declaration in the immediate scope, `a`. However, if it cannot be found in the immediate scope, the next outer scope (the parent scope of `a`) is consulted, continuing the hierarchy of ancestors up until the outermost (global) scope has been reached. In other words: A variable is valid in the scope it was created, as well as in all nested (descendant) scopes. This circumstance leads to the phenomenon of shadowing, which is described later in this chapter. As this way of looking up variables is executed *each time a variable is encountered*, it can have impacts on the performance as well, especially if the encountered variable is defined in a scope many levels higher in the scope chain.

```
function foo(a) {                          1
        var b = a * 2;                     2
        function bar(c) {                  3
                console.log( a, b, c );
        }
        bar(b * 3);
}
foo( 2 ); // 2, 4, 12
```

Figure 4.1: Nested scope in source code (Simpson 2014)

The manifestation of nested scope in source code is illustrated by figure 4.1. The function `foo` is defined *in* the global scope (1) (see next section), and is therefore accessible from all parts of this program. `foo` itself defines a new scope (2) which includes the identifiers `a`, `b` and `bar`. `bar` defines a new scope (3) within `foo`, defining only the identifier `c`. As can be seen, the innermost scope (3) has access to its own identifiers, as well as to the ones defined in its containing scope (2).

Figure 4.2 shows the scope hierarchy for the source code of listing 8.2 (see appendix). Assuming that, in a given context, the anonymous function nested inside `parseMarkdown()` is the active scope (marked in orange colour), the figure shows its scope chain, consisting of the three scopes `GLOBAL`, `parseMarkdown()`, and `(anonymous function)`.

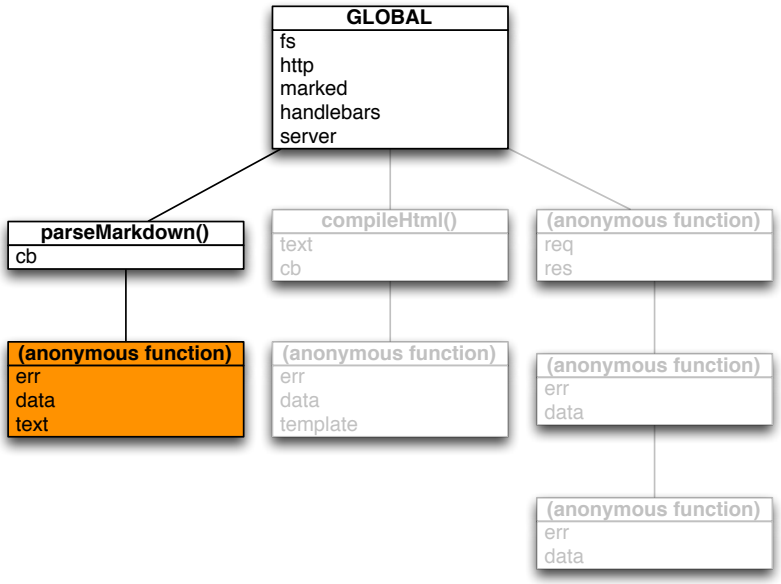

Figure 4.2: Scope hierarchy as hierarchical data structure; highlighted scope chain

4.2.2 Scoping Models

As mentioned above, the rules for when a new scope is defined differ depending on the programming language. Usually, a language implements multiple of the following rules.

Global scope Variables that are accessible from *any point* in the program are in the global scope. The original BASIC programming language only implemented global scope.

Block scope Any logical block, often denoted by containing curly braces (`{` and `}`), will create a new scope. This is the case in the C programming language, amongst others.

Function scope Any function definition defines a new scope. Parameters of the function are part of this newly defined scope, as well as variables and function defined within that function. JavaScript implements function scope.

Expression scope A variable's scope is limited to a single expression. This useful for very short-living, temporary variables. It is implemented by many functional languages, for example Python and ECMAScript 6.

JavaScript, as of ECMAScript 5, implements only global and function scope[1]. The run-time environment usually prepopulates the global scope with several objects and methods it exposes. The JavaScript engines in web browsers, for example, provide access to the Document Object Model (DOM) through the `document` object, whereas Node.js provides the `require` function to include CommonJS-style modules.

[1] There are exceptions through the keywords 'with' and 'except' and the function 'eval'.

4.2.3 Common scoping problems

The following are common phenomena that arise through scoping and may be the cause of problems and misconceptions. Though being typical for JavaScript, many of those problems can arise in other programming languages, in the same or similar form, as well.

These phenomena can in most cases be either helpful or hindering, and thus be desired or undesired. The goal of the concept developed in this thesis is to make the developer recognize those phenomena during author-time, and thus avoid misconceptions and reduce errors.

Hoisting is the implicit process, as done by the JavaScript engine, of moving variable and function declarations „from where they appear in the flow of the code to the top of the code“ (Simpson 2014). By code, Simpson refers to the scope block. Any variable declaration inside a scope block is hoisted to the top of the scope block.

```
function foo() {
  a = 2;
  var a;
  console.log( a );
}
```

The above code is actually processed as:

```
function foo() {
  var a;
  a = 2;
  console.log( a );
}
```

The variable declaration of `a` is moved, or „hoisted“, to the top of the scope block of `foo`. Hoisting can impose unexpected behaviour, especially when declaring variables of the same name in nested scopes.

Closure is a common phenomenon in JavaScript programs, and is widely used, though being generally seen as hard to understand. Citing Simpson, closure is „when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.“ (2014) As functions are first-class objects in JavaScript, they can be passed around like variables, for example as callbacks. A function can also return another function. However, JavaScript works with *lexical scope* and, according to the nesting rules presented before, a function must always have access to its ancestor scopes. Thus, when a function is being returned or passed as a callback, an instance of the whole scope chain is returned or passed along with the function. In other words, the function „closes“ or „forms a closure“ over its ancestor scopes. In most cases, this behaviour is desired.

Anyway it is important to recognize closures, as they may impact performance: the closed-over scopes have to stay in memory as long as a reference to the closure exists. Closure may also lead to unexpected behaviour, for example if a variable defined outside of a closure is used inside of it (see Simpson (2014, Ch. 5) for examples).

Shadowing is a consequence of nested scopes. If a variable (1) is defined in an ancestor scope, and a new variable (2) of the same name is defined in a descendant scope, the descendant scope has no access to (1). This is due to the mechanism of variable lookup explained above. Variable (1) is said to be *shadowed* by variable (2). As with most of the phenomenons listed here, this can either be desired or unwanted behaviour. A good solution to avoid shadowing is choosing different variable names throughout nested scopes.

Implicit variable declaration JavaScript allows for the creation of variables and object properties in an implicit way (*silently*). Instead of declaring a variable using a `var` statement, they can as well just be used without prior declaration, for example like this:

```
i = 3;
```

Variables used without prior declaration are implicitly declared in the *global scope*[1]. As this is usually unwanted behaviour, it is considered good practice to always declare variables explicitly. However, this problem is already addressed by linters (see 5.1: *Canonical Examples*).

Lookup performance The variable lookup through scope chains, as described above, can have an impact on the performance of an application. Each time a variable is encountered, the JavaScript engine performs the lookup process, navigating from the bottom of the scope chain upwards until it is found. If a variable, which is defined in an ancestor scope (the global scope, for example), is accessed within a deeply nested scope, the lookup process slows down the execution of the program, as shown by Castorina (2014). He furthermore suggests to cache the variable in a „closer“ scope, if possible.

Four of these five identified problems—*hoisting*, *closure*, *shadowing*, and *lookup performance*—need to be addressed by the design concepts created in chapter 5: *Ideation* and prototyped in chapter 6: *Design Iterations*. *Implicit variable declaration*, however, is already addressed by linting tools and is therefore not in focus of this design process.

[1] ECMAScript 5's *strict mode* considers this an error.

5 Ideation

This chapter exposes the ideation phase of the design process. By looking at canonical examples and solutions to conceptually similar problems, lessons are learned to support the ideation. A series of different concepts is sketched out, which ultimately leads to the design chosen for prototyping.

5.1 Canonical Examples

The most ubiquitous visualization of program structure is probably *syntax highlighting* or *syntax colouring*. This concept aims to make the developer distinguish entities of the programming language by showing them in different font types, weights, styles, or colours. According to the survey results (see section **??**: **??**), syntax highlighting can help with a number of different problems: recognizing errors and typos, distinguishing language constructs from variables and values, and orienting through specific visual patterns. Figure 5.1 shows syntax highlighting in an HTML document; HTML elements are printed in blue, whereas attributes are printed in purple, values in red, comments in yellow, and content in black.

In his talk „Monads and Gonads“, Crockford presents an alternative to syntax highlighting which he calls „context colouring“ (2013). Instead of using font styles and colours in order to highlight different elements of the *syntax*, he instead highlights different *scopes*. Figure 5.2 illustrates this concept: The global scope is presented in white, whereas nested scopes are marked green, yellow and blue, respectively. In this concrete example, identifiers are always coloured in the colour of the scope of *where they were defined*. For example, the appearance of `value` in the innermost scope is yellow, the colour of the scope in which `value` was declared (as a function parameter to the function `unit`).

Theseus is a JavaScript debugger built as a plug-in for the Brackets IDE. It makes use of the code editor itself and shows information inline, in the gutter and in a panel on the bottom of the Brackets window (see Figure 5.3). Theseus is mostly used for asynchronous debugging, so the way those UI elements are used corresponds to this purpose. For every function definition, Theseus shows the number of times the function has been called in the gutter. Functions that have never been called are marked with a grey background in the source code. Additionally, the panel on the bottom shows information about the function the cursor is positioned in[1].

[1] It shows asynchronous call stacks, which are not of relevance to this thesis.

```
1  <!DOCTYPE html PUBLIC "-//W3C/DTD HTML
2  <html>
3      <head>
4          <title>Example</title>
5          <link href="screen.css" rel="sty
6      </head>
7      <body>
8          <h1>
9              <a href="/">Header</a>
10         </h1>
11         <ul id="nav">
12             <li>
13                 <a href="one/">One</a>
14             </li>
15             <li>
16                 <a href="two/">Two</a>
17             </li>
```

Figure 5.1: Syntax highlighting in an HTML document

```
function MONAD() {
    return function unit(value) {
        var monad = Object.create(null);
        monad.bind = function (func) {
            return func(value);
        };
        return monad;
    };
}
```

Figure 5.2: Context colouring in JavaScript, as proposed by Crockford (2013)

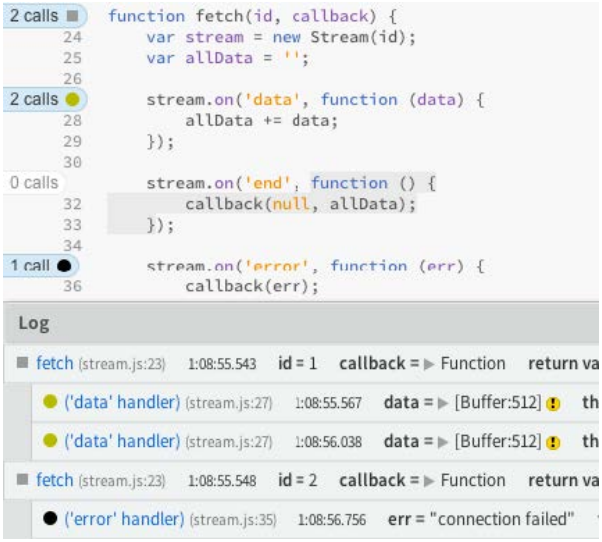

Figure 5.3: Theseus' asynchronous JavaScript debugging (Lieber et al. 2014)

JSHint is a so-called *linting* tool, or *linter*, for JavaScript: it detects bad programming practices (*code smells*) by checking JavaScript code against a set of rules, and therefore tries to prevent common problems. Originally built as a command-line tool and for online code checking, JSHint is implemented in many IDEs through the respective plug-in systems. The *Sublime Linter* plug-in[1] for Sublime Text 3 implements JSHint (and other linting tools) *inline*: hints of bad code or inconsistent style are shown in the text editor itself and are indicated in the gutter. If the cursor is on top of problematic code, the respective hint is printed in the status bar. *Sublime Linter* behaves according to the characteristics identified before: it is modular, as linters for different programming languages can be plugged-in; it is lightweight and does not slow the editor down; it focuses on code by displaying results inline without cluttering the editor window; and it is smart to some extent, as it allows the configuration of certain programming guidelines. The prototype presented later in this thesis borrows many characteristics and design decisions from linting tools such as this one. This is due to the fact that both the problem of code smells and the problem of scope can mostly be solved with static analysis[2] and presented in a similar manner.

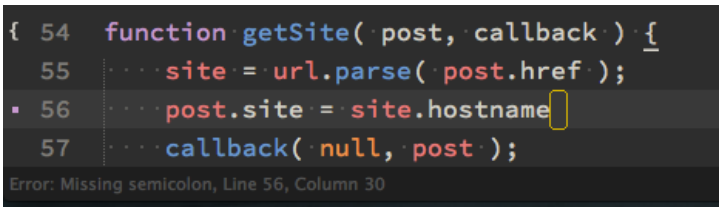

Figure 5.4: Sublime Linter indicating an error on line 56 (the semicolon is missing). The error is shown inline, in the gutter, and in the status bar.

[1] See `http://www.sublimelinter.com/`

[2] Static analysis is the analysis of computer software that is performed without actually executing the program.

In terms of navigating and displaying tree structures in relation to the actual source code, the *Element Inspector* of **Chrome Developer Tools** makes a good example (see Figure 5.5). The structure of HTML is quite similar to that of scope, as it is nested in the same way, which is why ideas can be taken from the Developer Tools. They show the source code of the inspected website and allows the user to select any HTML element within. In the remaining parts of the window, information relevant to the selected element is shown.

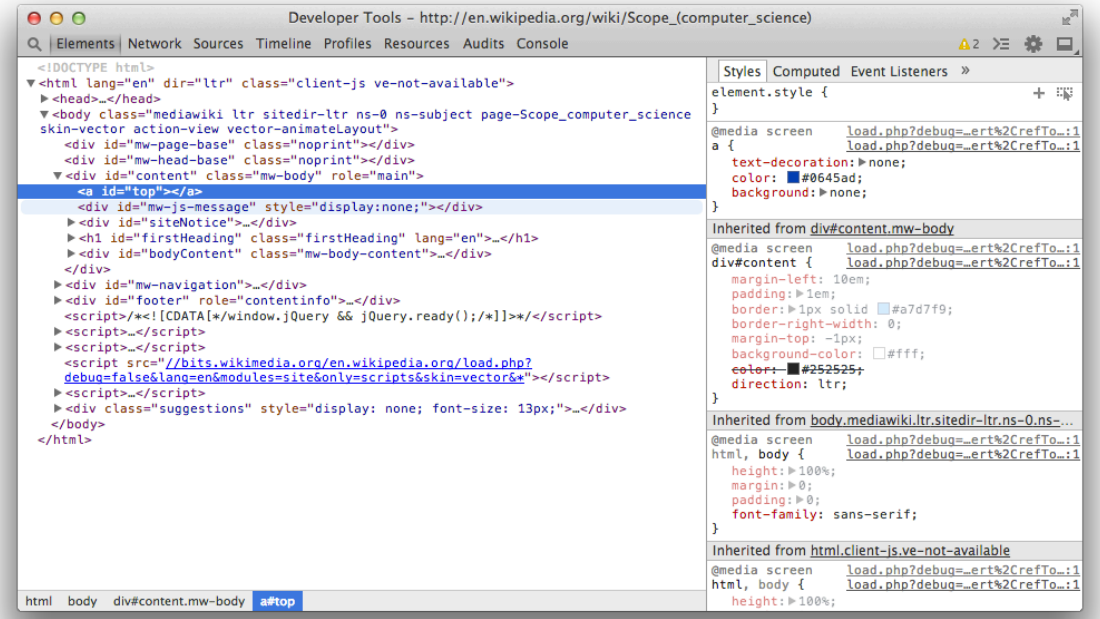

Figure 5.5: Chrome DevTools with Element Inspector

At the bottom of the window, a status bar shows the nesting of the selected element: on the visual left (and logical top of the tree) is the `html` element, inside it the `body`, then a `div` and finally the selected `a` element. This status bar can be used to navigate around the nested elements by clicking on them. Clicking on the `body` element highlights it in the source code as well, and shows different style information on the right-hand side.

Placed to the right of the source code is a sidebar. Although it contains a tabbed interface to browse different facets of the selected element, the one that is relevant—*Style*— is in focus on the screenshot. The way style (through Cascading Stylesheets (CSS)) is applied to HTML elements is similar to the way nested scope works: style that is defined on the containing elements may influence the style of the selected elements, which is why the relevant styles are listed in order of precedence. The style rules that apply with the highest precedence are listed on top, while the rules with the least precedence are listed on the bottom. Style rules that are overriden by rules of higher precedence are striked through to indicate that

they do not apply anymore. This way of visualizing and organizing information about nested structures is further used in the following concept and design phases (see section 5.2 and chapter 6).

5.2 Concept Generation

To support the ideation phase, existing UI components used within IDEs, as presented in chapter 2, were collected. Those components were written down on post-it notes and used as seeds for *seeded brainstorming*: for each of the components, a set of solutions should be developed that are similar, related to or based on the respective component. Most of the ideas that resulted from the brainstorming session made use of multiple components, for example the *scope chain* which is described further down: it made use of a status bar as well as the code editor.

Some ideas of the brainstorming phase made it into first sketches. Depending on if the idea depended on actual source code, the sketching took two different approaches. For ideas that involved code, it was important to work with real, functioning code. Therefore, two JavaScript applications were created to serve as examples:

- A small web server application, that parses a markdown-formatted text file and renders it into an HTML template. The application runs on Node.js and represents a typical control flow for e.g. a blogging engine (i.e. content + template = site).
- A client-side script (runs in a web browser) that fetches JSON data and presents them on a website, by the click of a button. This script represents typical client-side UI code, connecting a button event to a function and presenting the result in the UI.

The two applications were written in different styles: the server-side application decouples its tasks by putting them into specific, named functions (as far as it was seen appropriate) and therefore has a relatively flat tree structure. The client-side application, however, nests all function definitions inside each other, resulting in nearly one function definition on each line, and far deeper indentation (in other words, higher code complexity and deeply nested scope). A good solution for this design problem should address both of the cases.

Printouts of the two JavaScript files served as a basis for ideation that involved source code. However, for concept ideas that would mainly work with other UI components, such as a sidebar, or such concept ideas that would introduce new UI components, a blank notebook was used for sketching. The source code of the two JavaScript programs can be found in the appendix.

The following is a list of the concepts that have been assumed to be the most promising.

Scope Chain A breadcrumb view[1] of the active scope chain, similar to that of a selector chain in an HTML editor (see figure 5.5). The scope chain corresponding to the position of the cursor in the code would be displayed as breadcrumbs in a status bar. By hovering over a scope level breadcrumb, the corresponding source code of that scope would be highlighted in the source editor. Clicking on a scope level would navigate to the corresponding code, i.e. position the cursor to the top of the scope block.

Scope Graph Similar to a class browser, the scope graph represents a tree view of the application's scopes. The user can navigate the source code using the scope graph. It could be implemented as a sidebar or in a panel.

Scope Colouring Similarly suggested by Crockford (2013), the source code can be coloured depending on its scope level. Crockford's variation is meant to replace syntax highlighting; one could instead complement syntax highlighting by colouring the background (as Theseus does, see section 5.1: *Canonical Examples*), rather than the text—for example in different shades of grey.

Inspect Scope Comparable to the Chrome DevTools *Inspect Element* function, the user can right-click into the source code and choose „Inspect Scope", which opens a panel that shows global variables, current local variables as well as the value of `this` (the latter would be possible in a debugging environment, as `this` is only available at run-time).

Gutter Scope Any new scope created in the source code is indicated in the gutter. Comparable to how JSHint indicates errors (by placing a coloured dot or square in the gutter), the boundaries of the scope (i.e. its first and last line) are denoted by a coloured dot. For every scope level, the colour of the dot is different, so that the user can easily see how deeply nested the scope is at any given point.

Quick Inspect Similar to the *Quick Edit* feature of Brackets, the value of `this` can be inspected inline. By right-clicking on any point in the source code and choosing the context menu point „Quick Inspect", an area slides out inline (between two lines of source code), showing the value of `this` and available scope variables. As with the **Inspect Scope** concept, this would require to run in a debugging context to have access to `this`.

The concept chosen for prototyping is a combination of **scope chain**, **scope colouring**, and **inspect scope**. The *scope chain* is adapted as described above. *Scope colouring* is implemented in terms of background colouring. The *inspect scope* concept is adapted by means of its panel. The next chapter, Design Iterations, opens up the exact design decisions made for each of the elements.

[1] In the Yahoo Design Pattern Library: `https://developer.yahoo.com/ypatterns/navigation/breadcrumbs.html`

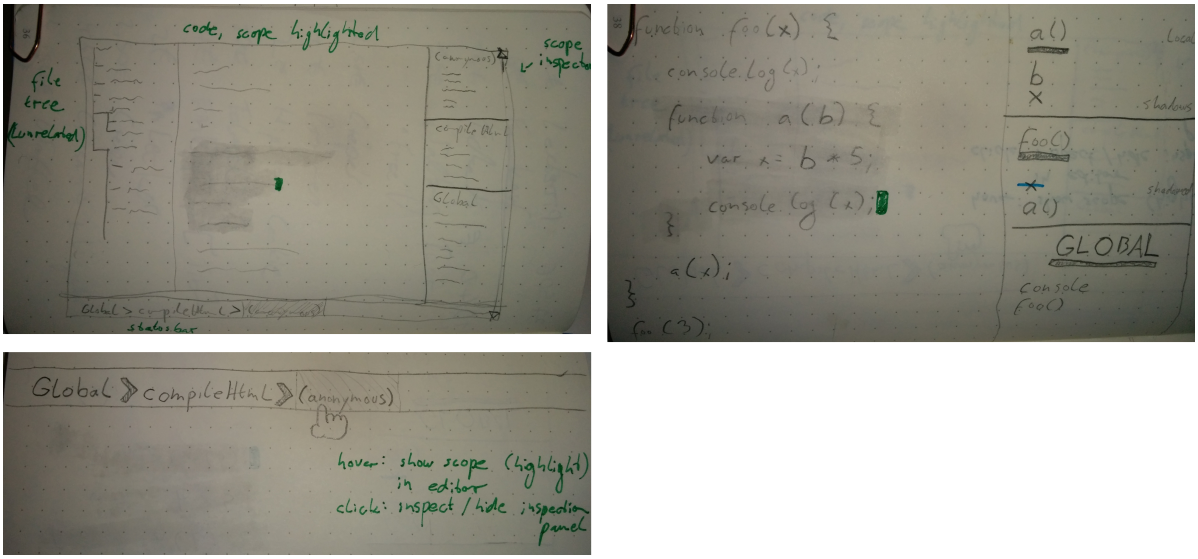

Figure 5.6: Some of the sketches created during ideation

6 Design Iterations

The following chapter exposes the design process and reasoning behind the design decisions made for the prototypes. It leads through the different prototyping stages and user testing, closing with an evaluation of the design.

The prototyping happened in three subsequent stages. The first stage comprised a set of pencil sketches on paper; the second prototype was created with web technology, but fixed around a certain source code and faked interactivity; the third prototype was implemented as a plug-in for the Atom editor, called *Scope Inspector*, and is able to work with any source code it is provided.

In chapter 3, we developed four **characteristics** of well-integrated language tools: performance, modularity, smartness, and a focus on code. Additionally, we listed five common scoping **problems** in chapter 4: hoisting, closure, shadowing, implicit variable declaration, and lookup performance. The design needs to address both the characteristics (non-functional requirements) and the problems (functional requirements). For every prototype, the addressed requirements are referred to in the respective section. However, some requirements are not addressed at all. The *smartness* characteristic is not explicitly focused on by the design, as in the case of scope there is no ambiguity. Neither is *Closure* addressed by any of the prototypes, for the reason that it is—from a technology perspective—hard to *correctly* detect. It is certainly possible , but given the short amount of time it was not a priority. Finally, linting tools already point out *implicit variable declaration*, which is why it is not in the scope of the design.

6.1 Definitions

To be able to discuss the qualitites of the concept and prototypes, we must first define a number of terms.

Scope block In a JavaScript text file, a scope block is the textual block representing a logical scope. For a function, which in JavaScript creates a new scope, the scope block starts at the `function` keyword and ends at the closing curly brace } of the function body. If, in a text editor, the cursor is placed anywhere inside this scope block (but outside of child scopes), the scope block is called *active scope*.

Current scope In a running JavaScript program, it is the currently executed scope. This is a term related to the run-time rather then to author-time, and should not be confused with *active scope* described below.

Active scope The scope which is currently in focus of editing. In relation to IDEs, code editors and the prototypes presented in this chapter, the active scope always describes the scope that the cursor is placed in.

Local scope In the context of nested scopes, the local scope is the one in focus (be it in the execution context during run-time, or the editing context during author-time). Local scope is contrasted with non-local scope; scopes that are logically distant from the local scope. Those may be ancestor scopes, descendant scopes, or parallel scopes. The term is also used to contrast *global scope*.

6.2 Sketching

One could argue that sketching is part of the earlier exploration phase, rather than of the prototyping phase. However, next to sketching different ideas, the author also sketched different possible implementations for one feature that seemed valuable to the design solution: *highlighting*. Highlighting of scopes through different colours—text or background colours—makes deeply nested scope more visible and thus directly addresses the problem of *lookup performance*.

The basis for the sketches were printouts of the same source code, each leading to a different way of highlighting.

Active scope, inclusive

This sketch highlights the active scope block by applying a background colour to it. The highlighting is *inclusive*, i.e. any descendant (inner) scopes are highlightes as well.

Active scope, exclusive

Same as above, but descendant scope blocks are excluded from highlighting. This way of highlighting was implemented in the scripted prototype (see section 6.4).

Active scope and ancestor scopes

Next to highlighting the active scope, its ancestor scopes can also be highlighted to emphasize nesting. To contrast the ancestor scopes from the active scope, the highlighting would make use of different background colours, for example different shades of grey. This way of highlighting can be combined with both the inclusive and exclusive approach.

Scope colouring

Described by Crockford (2013) as „context colouring“, this way of highlighting would not apply a background colour, but instead replace the existing forms of syntax highlighting. Thus, the highlighting would not depend on the cursor position (which defines the *active scope*), but would be static instead.

Identifier origin

Additionally to emphasizing code blocks, individual identifiers can be highlighted. Given a highlighted active scope, this sketch highlights identifiers that are defined in that scope but used somewhere else (in descendant scopes).

This works as well for the *scope colouring* described above, as each scope has a fixed colour. Identifiers that are used in other scopes than they are defined in can therefore always be recognized if they appear in the colour of their origin scope.

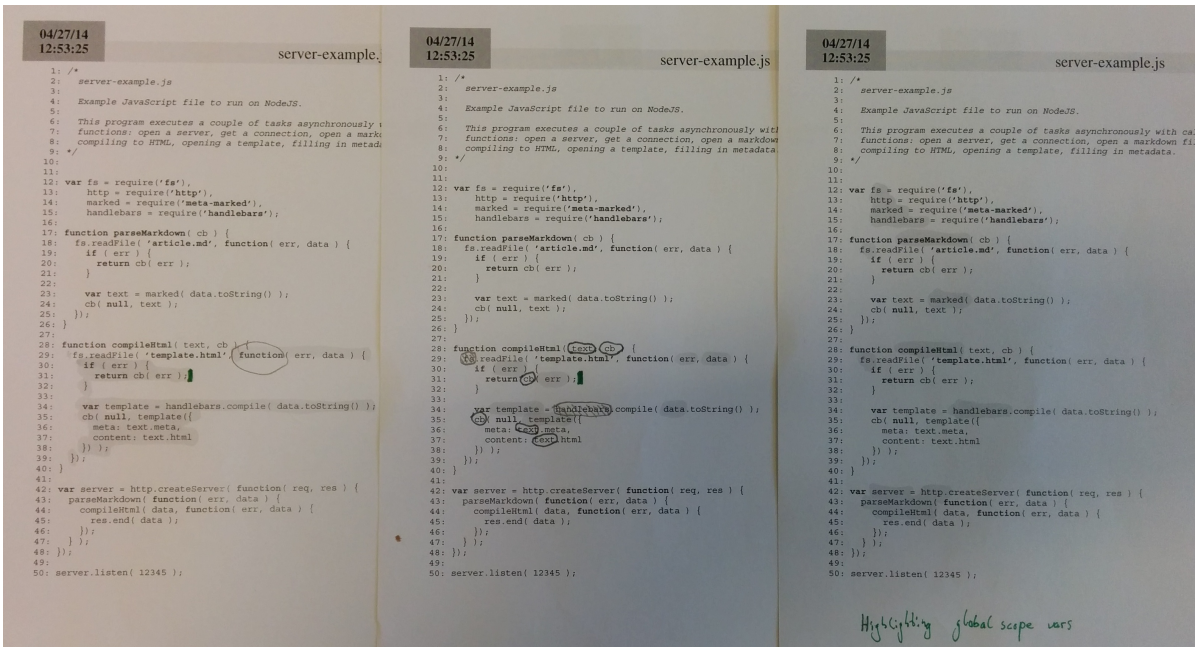

Figure 6.1: Some of the sketches to explore ways of highlighting

6.3 Scripted prototype

It very quickly became clear that the sketches were of little value. Although most of them gave a general impression on where the selected scope started and where it ended, it did not allow the user to see the big picture. It seemed probable that a more interactive prototype would be more helpful in this regard. As its capabilities of working with code are limited and the code had to be specifically prepared, this is called a *scripted prototype*.

As the author is most familiar with web technologies, the scripted prototypes would be built using Hypertext Markup Language (HTML), CSS and JavaScript and run in a web browser. Other prototyping tools, such as Balsamiq or Indigo Studio, would not allow for enough detail in terms of highlighting certain code passages, and would have represented a learning overhead.

Figure 6.2: Screenshot of the scripted prototype run in a browser

A code syntax highlighter[1] was used to turn the subject source code into styled HTML tags, to make it appear as if it was inside of a real code editor. Applying syntax highlighting was also necessary to see if the different highlighting techniques, as sketched out in the previous phase, would interfere with syntax highlighting.

Furthermore, markers in the form of HTML tags were added to the subject source code, which made it possible to apply different styles[2] to regions of the code. This was later used to realize highlighting of the scope block.

[1] Prism, see `http://prismjs.com/`

[2] For example text colours, background colours, of font styles.

Two distinct UI elements were added: a sidebar and a bottom bar. The content of both depends on the *active scope*, i.e. the scope the cursor is placed in. For each of the nested scope blocks that the cursor is positioned in (beginning from the local scope, going outwards up to the global scope), the sidebar shows a pane. Each pane contains the scope's name along with a list of identifiers defined within that scope. In opposition to the working prototype, the scripted prototype does not show phenomena like hoisting and shadowing. The panes are ordered ascending by logical distance, i.e. the local scope would be on top, the next surrounding scope beneath it, and so forth; up to the global scope on the bottom.

The different panes are hard-coded: all panes exist in the markup of the prototype at all times and are pre-filled with the relevant data, but are shown and hidden on demand.

The bottom bar shows a horizontal list of scope names. It makes use of the *breadcrumbs* UI pattern[1]. Each listed scope, beginning from the global scope on the very left, up to the local scope on the right, can be highlighted and navigated to by clicking on its label. By hovering[2] over the label, the user can get a preview of the target scope, as it is highlighted in the editor alongside the currently active scope.

6.3.1 Constraints

The scripted prototype has some drawbacks, some of which might influence its quality. The most obvious one is the fact that it only works with a static, predefined source code, which is manually adapted to serve the prototype's purpose. This implies that

1. changes can not be made to the code, which makes the experience of the prototype very different from a real code editor, and
2. it is hard to tell if the prototype works similarily well with code that is more complex, less complex, or of an overall different style.

The fact that there are no text editing facilities comes with another drawback, namely the absence of a cursor. If a cursor cannot be placed anywhere in the editor, the ‚activation' of a scope block must be achieved differently. In the case of this prototype, it is solved by clicking on a piece of code. However, clicking anywhere in the line besides the actual text will not change the active scope.

6.3.2 Evaluation of the scripted prototype

The prototype was tested with two JavaScript developers in individual in-person walkthrough sessions. The users were introduced to the concept, if they were not familiar with it already, and explained the basic constraints of the prototype (as mentioned above). They were thereafter able to explore and test the prototype to their likings. One of the two sessions have been recorded as a screencast.

[1] In the Yahoo pattern library: `https://developer.yahoo.com/ypatterns/navigation/breadcrumbs.html`
[2] Hovering: Placing the mouse cursor over an element.

The users liked both the „preview“ feature (hovering over a breadcrumb) and the sidebar. The preview gave them an opportunity to quickly get a visual overview of their position in the code and the active scope chain. The sidebar showed them which variables and functions are available in a given context. Overall, they liked the dynamicity of the prototype, as they could ‚play around‘ with it and understand the design concept just by trying. They quickly made a connection between the cursor position and the active scope along with its content.

One of the users suggested possible improvements or alternative designs for existing features. He recommended a wider use of colour coding to create a link between the scope in the editor window and the sidebar, for example by colouring all the ancestor scopes in different shades of grey. He also suggested an alternative visual structure for the sidebar instead of the list, for example nested clusters or a graph. For hoisting, the user came up with an idea to integrate indicators into the text editor: a „phantom“ variable declaration—which would be grey and not editable—could be inserted on top of a scope to show that a certain variable declaration would be hoisted up there. This indicator should be collapsible so that it does not interfere with the editing process. Because of technical constraints, this idea was not implemented in the next prototype iteration; however, it seems a sensible solution to the hoisting indication problem, as it communicates this implicit phenomenon very clearly.

In conclusion, the prototype was well-received and served its purpose well. It became clear that a consistent and clear visual language for the next iteration of the prototype was necessary, and that a direct connection between the code and the scope visualization has to be communicated. Like the sketches, it addressed the *lookup performance* problem by visualizing nested scope, even more so through the sidebar and bottom bar. It also put a stronger *focus on code* by making it navigatable using the bottom bar, and by dynamically showing changes directly in the editor.

6.4 Working prototype

The third and final prototype was built as a working prototype capable of handling any JavaScript scope, rather than as a proof-of-concept. It was integrated into the Atom[1] text editor as a so-called *package* or *plug-in*, released as Open Source Software (OSS) and was made publicly available for using and testing. The package is called „Scope Inspector“ and will be referred to using this name throughout this section.

6.4.1 The prototyping platform

For the prototype to yield meaningful results, I decided to integrate it into a real IDE. This decision was informed by several circumstances. The first and most obvious is that I could address a broader community of users this way. If the prototype was, like the scripted prototype, implemented in isolation

[1] See `https://atom.io/`

as a standalone application, it would raise the barrier for people to test and for me to distribute it. But by building it as a plug-in to an existing IDE, I could leverage the distribution channels that were already in place. A second reason is that users are already familiar with the software and do not have to orientate themselves anew. This also implies that all the features that the user *expects* from an IDE are in place, and the prototype can more seamlessly be integrated into the user's workflow. Finally, the third reason for building a prototype on top of an existing IDE is that it can make use of the design language in place, which eliminates the need to take decisions that are rather irrelevant for this prototype, such as the choice of a typeface and colour palette.

As a prototyping platform, the author decided on the Atom text editor. Atom is open source and created by the software company Github. By the time of writing, Atom is a relatively young project with a growing community and plug-in ecosystem. The reasons for deciding in favour of Atom are threefold: the technologies it is built upon, its internal software architecture, and the user group it is targeting.

Atom is built on web technologies, namely WebKit[1] and Node.js[2]. WebKit is the browser engine used by the web browsers Google Chrome and Apple Safari, amongst others, and is therefore responsible for the User Interface layer of Atom. Node.js is the JavaScript platform responsible for running any non-UI logic. Atom is mostly written in CoffeeScript[3]. Consequently, Atom packages can be written in CoffeeScript or JavaScript, using HTML and CSS for the UI. As I am familiar with these technologies, Atom provided an ideal prototyping platform with a low entry barrier.

For extending Atom, it offers an API which can be used by plug-ins. Atom's internal architecture is built in a modular way, so that plug-ins can hook into nearly everything that happens and react to it. The prototype makes use of this fact in many ways, for example by showing and hiding its UI elements depending on the type of file that is being edited.

Atom is marketed by Github as a „hackable text editor for the 21st Century"[4]. It is also intended to be a „deeply extensible system that blurs the distinction between ‚user' and ‚developer'." Those claims lead to the conclusion that Atom is a text editor built for developers, especially—but not exclusively—web developers. While not every web developer is a proficient JavaScript developer, the target groups of Atom and this prototype seem to overlap to a large extent.

6.4.2 Parsing and gathering relevant information

For the prototype to be as *complete* and *correct* as possible, it was built on top of an existing JavaScript parser called Esprima[5]. The process of extracting the relevant scope structure and annotations from the

1 See `http://www.webkit.org/`
2 See `http://nodejs.org/`
3 CoffeeScript is a programming language that transcompiles to JavaScript.
4 See `https://atom.io/`, accessed 18.05.2014
5 See http://esprima.org/

Abstract Syntax Tree (AST)[1] will not be discussed here in greater detail, but is instead described in a blog post (von Oldenburg 2014).

However, it is important to mention what data structures are extracted from the source code. Analogous to the nature of JavaScript scope as described in chapter 2: *Research Framework*, the data structure is a hierarchy of objects. Each object represents a scope and may have metadata as well as a list of identifiers attached to it. An identifier is either a child scope (as created by a function) or a variable. For scope objects, the metadata are its name and its location in the source code (row and column of the start and end points), whereas for variables the metadata are its name, location, if it is hoisted, by which child scope identifiers it is shadowed, and which identifier it is shadowing. Figure 6.3 shows the data structure by example of the source code in listing 8.2 (see appendix). The metadata are left out for the sake of readability.

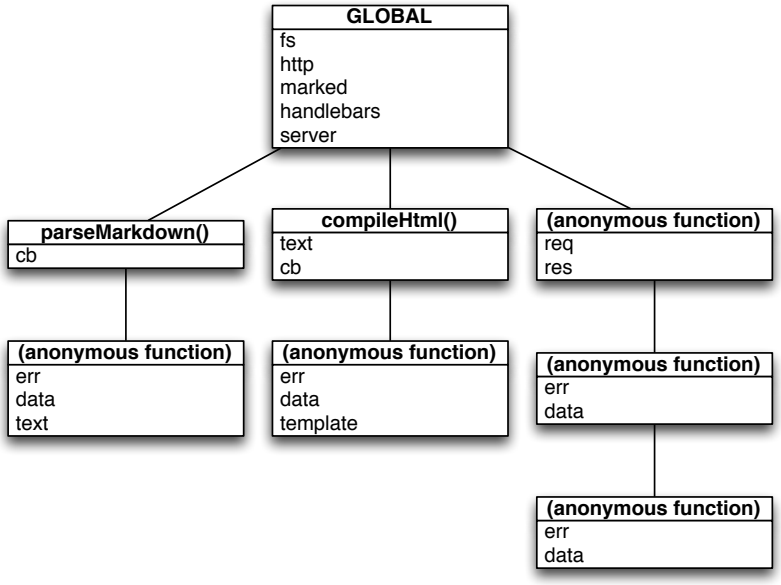

Figure 6.3: The data structure as utilized by the prototype

Using this data structure, the prototype can show meaningful data to the user, which would not have been possible with the AST alone. The modular composition of the prototype, which decouples the task of parsing from the task of displaying information, makes it possible to re-use each of the components. The component described in this section, which is responsible for turning the AST into a „scope tree“, could be used in plug-ins for any IDE to achieve similar functionality as the one of this prototype.

[1] The AST is the data structure that is returned by the parser, which contains all the lexical statements and expressions.

6.4.3 Interface and Interactions

The design respects that the subject of a developer's work is the code itself, not the tools that surround it. This is why the solution integrates into the most important part of the IDE, the text editor, directly. The features built into the editor itself will be called *inline* features.

Atom's interface is, by default, threefold: the text editor takes the most space; on its left is a sidebar containing a file browser, and on the bottom is a status bar. As many web browsers, text editors, and IDEs, multiple open files are accessed through *tabs* on the top of the screen. The tabs are important, because the Scope Inspector will only be active as long as an editor with a JavaScript file is in the foreground. Whenever the user switches to another tab, the Scope Inspector is activated or deactivated, depending on if the tab contains an editor with a JavaScript file or not.

The Scope Inspector package uses a visual style that is consistent with Atom's. Any icons in use are taken from the Octicons[1] icon set, which is incorporated into every Github product. Atom supports themes (colour schemes) for both the application window and the editor. Scope Inspector makes use of the colours defined in those themes. This way, the package UI feels more natural to the user. However, there may be difficulties if the theme is not well-defined and the colours are badly balanced. One user reported very low contrast between the editor's background and the scope highlighting. In addition to pre-defined theme colours, Atom also provides a set of pre-styled UI components, for example buttons and panes, which have been used in the prototype.

Whenever the Scope Inspector is active, two things are obvious: on the bottom of the editor, a panel is shown which we call *bottom bar*, and the active scope is highlighted inline. Additionally, a sidebar can be toggled using the Atom command „Scope Inspector: Toggle Sidebar". This command is accessible using the menu, the command palette, a keyboard shortcut (`Ctrl+Alt+i` by default), and a toggle button on the bottom bar. The user can choose wether or not to use the package with the sidebar enabled, which addresses the *modularity* characteristic. For *performance* reasons, the JavaScript program is not re-evaluated (i.e. its scope structure is built up) on every character that is typed by the user, but only when its file is saved. The several components and their functionality are explained in more detail in the following sections.

Inline scope highlighting

As explained above, the *active scope* is the immediate scope the cursor is placed in. It is emphasized by highlighting it through a lighter or darker background colour (depending on Atom's colour scheme). If the cursor is placed in a different scope, the formerly active scope is de-highlighted, and the now active scope is highlighted instead.

[1] See `https://github.com/styleguide/css/7.0`

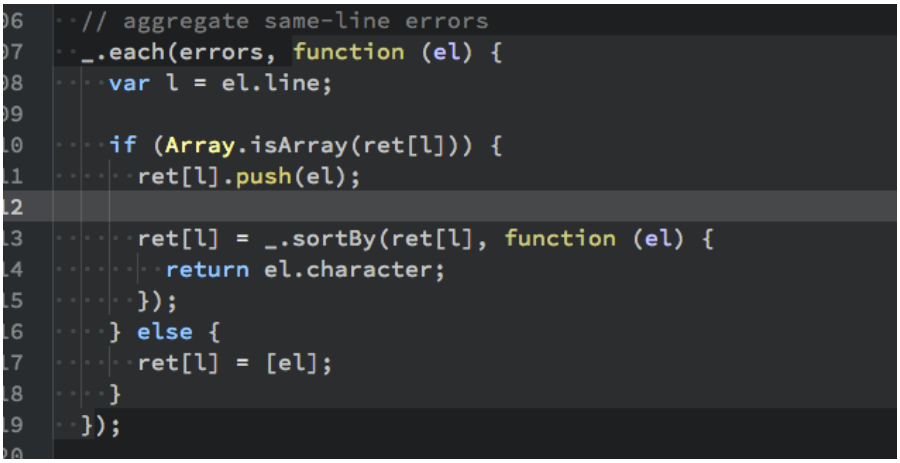

Figure 6.4: Subtle highlighting of an anonymous function

While the scripted prototype implements *exclusive highlighting*, the working prototype implements *inclusive highlighting*, which means that the inner scopes are highlighted as well. This is due to technical reasons; building exclusive highlighting into the prototype would have taked a lot more time. In further iterations of the prototype, an option to enable and disable exclusive highlighting could be provided.

The bottom bar contains a toggle button[1] to enable or disable highlighting of the global scope. Highlighting the global scope with *inclusive highlighting* is not useful, as the whole file would be highlighted (and there would be nothing left to contrast the highlight to).

Inline hosting indication

If there are identifiers being hoisted in the active scope, an indicator—which consists of a dotted line with an arrow head—is shown inline. The indicator appears at the place in the source code where the identifiers are hoisted to. This is *before* the first statement in the given scope which is not a variable declaration. In figure **??**, this is before the very first line of the scope block. By hovering over the indicator line, the user reveals a tooltip listing all the identifiers that are being hoisted to this place. This feature obviously addresses the *hoisting* problem, and does so while maintaining a *focus on code*.

Figure 6.5: Inline hoisting indicator with tooltip

[1] A switch in the form of a button, which can be either *on* or *off*.

Bottom bar

The bottom bar serves two purposes: it provides a quick glance of where in the scope hierarchy the cursor is and provides quick access to two settings.

Figure 6.6: The bottom bar, showing the scope chain and toggle buttons

On the right side of the bottom bar, two toggle buttons allow for enabling and disabling of two features. The right button, showing a list icon, shows or hides the sidebar. The left button with the label „Highlight Global“ toggles the highlighting of the global scope (as described above).

The left side of the bottom bar shows the breadcrumbs known from the scripted prototype. The breadcrumbs, implemented as simple buttons, are labeled with the corresponding scope name. The global scope is always on the left, whereas the currently local, active scope is on the right. By hovering over any of the breadcrumb buttons, the user can preview the respective scope highlighting in the editor. The preview is applied in addition to the currently active highlight in a different colour.

By hovering over the breadcrumbs from left to right or from right to left, the user can make the relationship between the logical structure of the JavaScript program (in the form of hierarchic scopes) and the textual structure (in the form of code) visible. As in the previous prototype, the bottom bar emphasizes the scope nesting and thus addresses the *lookup performance* problem.

Sidebar

The sidebar shows content depending on the currently active scope. Similarily to the scripted prototype, the sidebar lists one pane for each scope in the hierarchy of the active scope. The active scope is listed on top, while its ancestors are listed below, up to the global scope on the very bottom.

Figure 6.7: Sidebar (clipped), with shadowing, shadowed, and hoisted identifiers

Each pane is entitled by the name of the scope. In case of function scope, the name of the function becomes the scope name („(anonymous function)“ in the case of an unnamed function expression). In case of the global scope, the name is „GLOBAL“. Underneath the title, the names of all identifiers defined within the scope are listed, along with certain attribute annotations.

- Function parameters are listed first. They appear with the annotation „param“, set in smaller text size to the right.
- General variables follow the parameters. If they are not shadowed, they have no annotations.
- Functions are the last entities in the list. They are connotated with a pair of parantheses „()“.

The listed identifiers show also if they are hoisted, shadowed, or if they shadow other identifiers. This is indicated by different stylistic changes.

- Hoisted identifiers have a small, upwards-pointed arrow on the left side of their label. This indicates that their declaration is implicility moved upwards in code.
- Shadowed identifiers are printed in a more subtle text colour. Besides that, their label is striked-through to indicate that they are not accessible within the given descendant scope.
- Identifiers that shadow other identifiers in ancestor scopes are printed in a highlight colour. In case of Atom's standard UI theme, this is a bright blue colour.

Consequently, the sidebar addresses both the *shadowing* and *hoisting* problems.

6.5 User Testing & Evaluation

The goal of user testing was to collect both qualitative and quantitative data through different methods. The quantitative data collection was built into the prototype in form of a connection with Google Analytics[1].

6.5.1 Test installment

Atom includes a package management system with an online repository, called Atom Package Manager (APM). This system allows developers to publish Atom packages and thus make them available for any Atom user to download and use. Consequently, this prototype was distributed via APM.

I originally recruited two full-time developers and one part-time developer for remote user testing. They agreed to install the package and use it over the course of one week (full-time developers) or one day (part-time developer), respectively, integrating it into their usual workflows. I would subsequently conduct interviews with them, as contextual inquiries were not possible, due to the fact that the developers are located in Germany and I am located in Sweden. However, neither of them was available for the actual testing phase. The full-time developers were—during the relevant time span—working on tasks of their job that do not involve JavaScript development, whereas the part-time developer did not have any relevant work at all.

Instead, I relied on local and remote contextual interviews. Four potential users agreed to test the prototype in short sessions (30-60 minutes). The sessions were not directed, except from a short introduction to make sure the user knows about the research background and the *scope* concept. The users were asked to speak out loud during the session, and I was helping them if they got stuck or did not discover a feature. Two of the four sessions were recorded using screencasts; the others were not recorded due to technical reasons. Two of the four test users had previously been interviewed during the research phase.

In addition to this planned user test, publishing the prototype via APM made it available to the general public. It was announced on several social networks, especially targeting existing Atom users, with the goal of getting users to download and use it. Two weeks after publishing the prototype, the number of downloads counted ~150. This way of testing „in the wild“ makes it harder to gather feedback, compared to the method of addressing potential users directly. However, it also yielded unstructured, but valuable, feedback on several social media channels.

[1] A website and app analytics platform.

6.5.2 Usage metrics

The prototype was built with the option to collect usage metrics using the Google Analytics service. By default, this option was set to *off*, as I oppose the unknown tracking of any data for ethical reasons. However, users were asked in the `README` file, which is displayed both on the package's page[1] and on Github[2], to enable tracking in Atom's *settings* panel for the Scope Inspector package.

If enabled, the following events are tracked:

- The package is enabled/disabled
- The sidebar is shown/hidden
- The user hovers over a scope breadcrumb in the bottom bar and thus previews a scope highlighting
- The user clicks on a scope breadcrumb in the bottom bar and thus jumps to the beginning of scope, making it the active scope and highlighting it

Through collecting these events, a claim can be made—to a certain extent—for how helpful certain features are.

6.5.3 Testing Results

Analytics

Analytical metrics have been tracked over the course of 13 days. In total, five different users (including myself) had tracking enabled, with a peak of fours simultaneous users. On average, two to three users contributed data each day.

In those nearly two weeks, the sidebar was enabled 76 times, whereas interactions with elements on the bottom bar happened 144 times. However, only 22 of those interactions were clicks which have to happen intentionally—the hover events can occur by chance (for example by moving the mouse from the editor window to the status bar and vice-versa). Given those circumstances, it can be concluded that the sidebar was more in use than the bottom bar. But event metrics can not measure if users used the bottom bar for orientation, for example just by looking at it and figuring out where in the scope hierarchy they are. Thus, the analytics result are of limited use for the evaluation of the prototype.

Interviews with developers

This section summarizes the finding of the user testing, as collected through interviews. None of the interviewees had used the prototype before, and both are only casual Atom users.

[1] See `https://atom.io/packages/scope-inspector`
[2] See `https://github.com/tvooo/scope-inspector`

Inline The inline highlighting was perceived as useful, as users claimed it helped them to focus on the current code and also showed them the scope limits (in code). None of the developers discovered the inline hoisting indicator, it had to be pointed out to them. Even after pointing it out, the users did not use the inline indicator, but instead just looked at the sidebar to see which identifiers were being hoisted. One of the developers was not familiar with the concept of hoisting.

Sidebar One functionality that each user found to be missing was a means of navigation through the sidebar. Two users expected to be able to jump to a variable declaration if he sees a problem there, just by clicking on the variable's name. Another user thought the ability to scroll to a certain variable was denoted by the upwards pointing arrow next to it. It was also suggested to be able to jump to the beginning of a scope by clicking on its respective headline, similar to what happens when the user clicks a breadcrumb in the bottom bar. Adding line numbers for each identifiers was also suggested, in order to be able to distinguish anonymous functions better from each other.

The order of the panels, each representing one scope in the scope chain, was confusing to some users. Although they understood the concept of a scope chain, they expected it to be in the order that scopes appear in code. However, code is linear, while scope is hierarchical, and they do not map directly. It is reasonable to assume that the order which is used in the prototype is learnable for the users, as it works analogously to CSS in the Chrome DevTools. This could for example be achieved by connecting the scope in the sidebar visually with its counterpart in the editor. One user suggested that, when a scope is hovered in the sidebar, its counterpart in the editor could be highlighted (analogous to the effect of hovering a breadcrumb in the bottom bar).

Another suggestions concerning the sidebar was to highlight even single variables in the editor when they are highlighted in the sidebar. Regarding shadowing, one user was able to detect a bad practice in his code during testing: he had created a shadowing situation in which the two variables of the same name server completely different purposes. Another user misinterpreted the hoisting indicators (upwards arrows) in the sidebar. He assumed that they are pointing upwards because clicking on them would cause the editor to scroll upwards, navigating to the identifier's declaration.

Bottom bar None of the users discovered on their own the possibility to navigate the scope chain by clicking on the breadcrumbs. However, after pointing the feature out, they stated it was useful. One user noted that, once you navigate into a higher scope, you can not go „back“ to the last position (or the previously selected scope, which is nested inside the now active scope).

Two users stated that the bottom bar would take up too much space, as most developers would like as much space as possible for their code. They suggested to style the breadcrumbs more like hyperlinks on the web, and also to indicate that they are breadcrumbs by deparating them through rightwards pointed arrows. One user even suggested to get rid of the bottom bar completely, in case that the sidebar would take its tasks of navigating and previewing the scope chain.

Modularity One of the users asked if he could disable the inline highlighting altogether, as he could imagine it to be annoying in the long run; this is not possible in the final prototype, as during the design phase the highlighting was considered a central element of the design. He suggested that all three parts of the prototype—inline highlighting, sidebar, and bottom bar—should be optional.

Miscellaneous Various comments concerned none of the areas above. A suggested feature was for the prototype to give an overview of all the identifiers that are affected by shadowing or hoisting, in order to quickly uncover possible code smells. In general, so thought one user, would it be a good idea to communicate the *meaning* of a detected code smell and show them how to interpret it: Why is it a possible defect, and what can the developer do to fix it? This would be especially useful for more complex findings, such as closures (which are not implemented in this prototype). Those suggestions drive the design more into an educative direction. Additionally, one bug was found during testing, which is related to the parser and therefore not relevant to the interface.

In total, the prototype did well in user testing. Users stated that it was unobtrusive and helpful, especially the highlighting and sidebar. However, the bottom bar was not as useful to the developers, which mirrors the quantitative results.

Social Media Feedback

In the open source software community, social media channels are frequently used to state an opinion on new products, to share news, and to give feedback. The prototype, in the form of the Scope Inspector package for Atom, received positive feedback on different channels.

Several reactions[1] on the blog post about the technical side of the Scope Inspector (von Oldenburg 2014) suggest that there are difficulties for JavaScript developers to deal with CoffeeScript code. In other words, the target users of the package (JavaScript developers) are not directly able to improve on or modify the package itself. This complicates code contribution for this open source project.

On Github, users filed two issues (bugs). One user requested the possibility to investigate identifiers in the sidebar more deeply (which is impossible to do correctly if the program is not executed). The other one asked for on-the-fly re-evaluation of the scope, as was already asked for in the interviews. At the stage of the prototype, scope is only re-evaluated when the file is saved; looking at linting tools, re-evaluating a short time (100-200ms) after the user stopped typing is a proven strategy[2]. However, this method needs to be tested in terms of performance.

[1] See http://www.echojs.com/comment/9867/1 and https://twitter.com/vilmosioo/status/466934333681717248

[2] See https://github.com/tvooo/scope-inspector/issues/7

On the news platforms EchoJS and Reddit (JavaScript section) the package got little attention: four and eight upvotes, respectively, although the only comment on Reddit claims that it „looks promising“[1].

On Twitter, which was used as a marketing instrument as well, the feedback was exclusively positive. One user called it „another reason to consider switching from [Sublime Text to Atom]“[2], while others seemed to be interested as well[3]. Five tweets mentioning the package have been both retweeted and favourited about 20 times[4].

[1] See http://www.reddit.com/r/javascript/comments/25k30l/javascript_scope_inspector_an_atom_package_to/chi27cm

[2] See https://twitter.com/adardesign/status/466589449561055232

[3] See https://twitter.com/raganwald/status/466930480517246976

[4] Based on tweets that could be found by the Twitter search for the keyword “‘scope inspector’”.

7 Discussion & Reflection

The following chapter discusses the results of the user testing (evaluation) with respect to the goals presented in the introduction. It also reflects on the design process and makes suggestions for improvements in both the process and the concept.

7.1 Quantitative reflection

From a quantitative standpoint, it is hard to tell if the *Scope Inspector* package is useful or successful. The following two metrics are taken into consideration.

Number of downloads The number of downloads, as presented by the package website[1], is a metric that has to be compared to others to drive a meaningful conclusion. The total number of Atom installations is unknown to the author, however, it is possible to make an estimate: The package *Find and Replace*[2] is part of Atom's standard distribution and thus should have been downloaded about the same amount of times[3]. By the time of writing, *Find and Replace* has been downloaded 14.000 times. Compared to this, the number of downloads for *Scope Inspector* seems low, however, one has to take different circumstances into consideration:

1. The Atom editor, and thus the *Find and Replace* package, has been around much longer than the *Scope Inspector* package. The first release is dated to September 17, 2013[4]. Thus, the package has been around for about 8 months, while this prototype has only been around for two weeks by the time of writing.
2. The *Find and Replace* package provides a core functionality of every text editor, whereas the *Scope Inspector* package fulfills a specialized task for a narrow target group.
3. The *Find and Replace* package comes with each Atom installation. However, to even *get to know* about the Scope Inspector package, one has to search for stumble upon it.

[1] See `https://atom.io/packages/scope-inspector`
[2] https://atom.io/packages/find-and-replace
[3] This estimate does not include custom builds of Atom, for example by compilation of the source on a Windows or Linux platform.
[4] See `https://github.com/atom/find-and-replace/tree/v0.2.0`

Based on these arguments, one could find a package with more similar characteristics to do a comparison. The one that comes closest in terms of target group and functionality is the JSHint plug-in for Linter package, which provides a JavaScript linting tool[1]. From its first release on April 18 until the time of writing, the package had been installed about 4900 times, which is about 30 times the number of downloads of Scope Inspector in twice the time. Given the familiarity of linting tools, which are part of most JavaScript developers' workflows, and the novelty of Scope Inspector, one can conclude that the Scope Inspector's number of downloads is moderately high. However, while the number of downloads may proof that there exists some interest in this sort of language tool, it is not a sign for its usefulness.

Analytics Including me, analytics have only been enabled by five users. There may be a couple of reasons for this low number.

1. Analytics tracking is disabled by default for ethical reasons: I do not want to track users without them knowing and explicitly giving their consent.
2. Users do now want to opt-in to analytics tracking, or consider it to be unimportant. In this case, a stronger point for the benefits of tracking for both the user and me should be made.
3. The package is only used by a very low number of users. This is likely, but the difference between the five users who enabled analytics and the 150 who downloaded it seems to be too high—the real number of users is probably in between 5 and 150.
4. The request to opt-in is too subtle and not visible enough. It is quite possible that users do not read the README file thorougly[2]. In this case, the option to opt-in has to be made more prominent. It can be moved up in the README file and emphasized better. Another approach would be to advertise the tracking (along with this research project) in the package-itself, for example by showing an information window on first activation of the package.
5. The tracking code is broken. This is not very possible, as in this case, the analytics tool would not have collected any data. However, it collected data throughout the two week testing period.

It is probable that opt-in by default would have raised the number of tracked users significantly. However, as I stated before, this would be strongly against my ethical position and I am convinced that there are other ways to gain relevant testing data. The next most promising approach is to raise the visibility of the opt-in option and market the research better, as stated above in (2) and (4).

With the low number of analytics opt-ins, it is hard to draw a conclusion on the usefulness of the package and its single components. However, as the sidebar was being enabled each day of the

[1] See `https://github.com/AtomLinter/linter-jshint/tree/v0.0.1`

[2] The README file is commonly used in open source projects to communicate a project's purpose and even document it. Against many presumptions, it is also usually read.

testing period with the exception of one, it can be claimed that the package was actively used by all of the opted-in users, and not just activated but unused.

7.2 Qualitative reflection

More meaningful than the quantitative metrics are qualitative results. The original goal was to evaluate the prototype by distributing it to the users and let two developers use it for at least one week in a real working environment. However, as mentioned in the previous chapter, the recruited developers were unable to use the prototype in their daily work, because they were mainly developing with HTML and CSS during that time period. For that reason, I had to rely on other ways of testing: remote interviews and unstructured feedback via social media channels.

Remote interviews The testing scenario was not optimal due to the following circumstances:

1. All users except one were only available for remote testing. The one available locally was using JavaScript only casually and not on a daily basis. It would have certainly been more fruitful to work with professional developers locally in Malmö. However, I failed to recruit developers in the short period of time available—especially because of the narrow target group of the concept—and thus relied mostly on developers I knew from previous work in Germany. With a less restricting time limit, this circumstance could most possibly be improved upon.
2. None of the users who was available for an interview was able to test the prototype during a whole week in a real working environment. Instead, only in situ testing with prepared and existing source code could be conducted. Given more time, it might have been possible to recruit more full-time developers. Also, the two developers already recruited could have participated in testing at a later point in time.

Given the unfortunate circumstances, the feedback gathered through remote interviews is detailed and useful. It can be used as a solid grounding to enhance the prototype in future iterations. The only point where the feedback comes short is in long-term testing—the question how well the prototype can be integrated into existing development workflows cannot be answered satisfactorily.

Social Media Channels Social Media have been used marketing as well as feedback channels. Announcing the Scope Inspector on Twitter, EchoJS and Reddit contributed significantly to its distribution. The feedback yielded on Twitter and Github—exclusively from users who are unknown to me—shows that social media are suitable channels to communicate about and announce open source projects. For future projects it will be a good idea to ask influences on those media—JavaScript experts with a lot of connections—to look at and write about it. This way, even more traction could be gained.

Atom—along with APM—is a new platform with a growing community, which is still in its early phase. The impacts of these circumstances have been described in detail in this chapter. Whereas I expected the social media coverage to be lower than it was, the actual user testing did not go without problems.

For similar projects in the future, it thus seems best to choose a more mature platform—one with more users, available on more operating systems, and with an extensive ecosystem of plug-ins, knowledge, and community. A good example for a project on such a platform is Theseus by Lieber et al. (2014). Based on Brackets, which is around way longer than Atom and more widely distributed (comparing the activity on Github), it had a better chance and more time to be adapted. On the other hand, the momentum of growing platforms like Atom—especially if it is backed by one of the biggest open source communities (Github)—should not be underestimated. The Scope Inspector will most likely benefit from this in the future.

8 Conclusion & Outlook

While the disciplines of Interaction Design and Software Development are deeply intertangled, the world of IDEs and programming language tools is still mainly driven by technology. This thesis made an approach to apply interaction design methodology to this area by designing a language tool for a very specific, narrow target group.

8.1 Knowledge Contribution

Through the design process, this thesis contributes knowledge to the interaction design community. Four characteristics of well-integrated programming language tools—performance, modularity, smartness, and focus on code—have been identified. Those characteristics can be used as a loose grounding to evaluate language tools for professional software developers. Other target groups may prioritize different characteristics.

They way that the user testing was planned and prepared seemed, at that time, reasonable. Due to my background in web development, I recruited professional developers I knew for user testing. However, the fact they could not make use of the prototype due to work reasons was unforeseeable for me. The subsequent attempt to recruit users through the survey failed as well—most users knew JavaScript to some extent, but for none was it the primary professional language. Given the fact that users would need to work with Atom, which only runs on the OS X platform by the time of writing, problems with user testing could have been expected. Apparently, testing prototypes with a very limited user group in the open source community implies that great efforts have to be taken to recruit users, and longer time periods are necessary than have been given in this case. However, given enough time and access to more test users, this approach of evaluating designs in the field—with analytics, long-term tests, and social media—is promising.

The way that the prototype was put into the world, marketed and evaluated seemed to be reasonable for an open source project and this target group. The lesson learned was that the short amount of time available for this project makes long-term testing extremely difficult, especially if the target group is so narrow. Given a longer time span and access to more test users, this seems to be an efficient way of evaluating designs like this.

Picking *scope* as a topic of focus, I was quite surprised that there could no integrated language tool be found that addressed scope in the way my design does—especially as the survey and interviews identified it easily as an ongoing issue in programming. This leads to the assumption that user-centered design methods, as were applied in this project, do not usually lead innovation in open source projects like there. A quick look into several open source projects[1] of different size shows that user participation exists in such projects, but is realized in a much looser way, for example through whole ecosystems of institutions and communities, as in the case of Linux (Raymond 1999); through suggestion platforms, as in the case of Mozilla[2]; or through open contribution, as in the case of the masses of smaller open source projects and made possible through platforms like Github and Launchpad[3]. Raymond (1999) states that every good work of software starts by scratching a developer's personal itch. However, the user-centered design process forces the designer to look for *other developers' itches*. Thus I think that the result of this thesis differ from what a classical open source approach may have resulted in. Applying interaction design methodologies prooved to be highly beneficial, and will probably be so for other open source projects. While this knowledge contribution was not anticipated, it is very welcome.

8.2 Applicability

Scope is a concept that appears in most programming languages. While JavaScript was taken as an example throughout the design process, the design is transferable to other languages as well. Closest to JavaScript are, of course, other languages that implement functional paradigms and similar scoping rules, for example Clojure, Dart, or Python. However, even originally non-functional languages are starting to implement features like closures, most recently Java and PHP. But even without functional features, scope remains a problem for developers of many programming languages, beginners and professionals alike. The concept created in the course of this thesis can be beneficial to them, as well.

Additionally, the concept is quite independent from the development environment. Though it was using the visual language and interaction models of Atom, those can be as easily adapted to IDEs like Eclipse, Visual Studio, or IntelliJ as they can be to Sublime Text, Vim, or Emacs.

8.3 Outlook

The feedback gathered through user testing is most helpful in driving the Scope Inspector forward. It will be interesting to see how it will be adapted by the community of JavaScript developers as it starts

[1] User-driven innovation processes in open source software are subject to a whole other line of research, and are thus not discussed in detail here.

[2] See for example: `https://hacks.mozilla.org/2014/05/developer-tools-feedback-channels-one-week-in/`

[3] See `https://launchpad.net/`

to implement more relevant features, such as closure detection and sidebar navigation. More time will allow the long-term testing that this thesis project could not provide, and the prototype's value will be proven or disproven in the long run. In the meantime, it will be interesting to see if and how interaction design methodology drives innovation in the open source community.

Bibliography

Baxter-Reynolds, M. (2011), 'Program in a text editor rather than an IDE? Why would you do that?'. Accessed: 22.04.2014.
URL: *http://www.theguardian.com/technology/blog/2011/oct/24/programming-ide-editors-choice*

Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. & LaViola Jr., J. J. (2010), Code Bubbles: A Working Set-b ased Interface for Code Understanding and Maintenance, *in* 'Proceedings of the SIGCHI Conference on Human Factors in Computing Systems', CHI '10, ACM, New York, NY, USA, pp. 2503–2512.

Castorina, D. (2014), 'JavaScript Prototypes, Scopes, and Performance: What You Need to Know'. Accessed: 16.05.2014.
URL: *http://www.toptal.com/javascript/javascript-prototypes-scopes-and-performance-what-you-need-to-know*

Crockford, D. (2013), 'Monads and Gonads'. Accessed: 26.04.2014.
URL: *https://www.youtube.com/watch?v=b0EF0VTs9Dc*

DeLine, R., Czerwinski, M., Meyers, B., Venolia, G., Drucker, S. & Robertson, G. (2006), Code Thumbnails: Using Spatial Memory to Navigate Source Code, *in* 'Proceedings of the Visual Languages and Human-Centric Computing', VLHCC '06, pp. 11–18.

Granger, C. (2012), 'Light Table — a new IDE concept'. Accessed: 07.04.2014.
URL: *http://www.chris-granger.com/2012/04/12/light-table—a-new-ide-concept/*

Hidayat, A. (2012), 'Language Tools for Reducing Mistakes'. Accessed: 03.05.2014.
URL: *http://ariya.ofilabs.com/2012/11/language-tools-for-reducing-mistakes.html*

Interaktives Programmieren als System-Schlager (1975). German. Accessed: 28.04.2014.
URL: *http://www.computerwoche.de/heftarchiv/1975/47/1205421/*

Johnson, B., Song, Y., Murphy-Hill, E. & Bowdidge, R. (2013), Why Don't Software Developers Use Static Analysis Tools to Find Bugs?, *in* 'Proceedings of the 2013 International Conference on Software Engineering', ICSE '13, IEEE Press, Piscataway, NJ, USA, pp. 672–681.

Lieber, T., Brandt, J. & Miller, R. C. (2014), Addressing Misconceptions About Code with Always-on Programming Visualizations, *in* 'Proceedings of the SIGCHI Conference on Human Factors in Computing Systems', CHI '14, ACM, New York, NY, USA, pp. 2481–2490.

Lynch, D. (2011), 'Why not just use an IDE if you want IDE features?'. Accessed: 22.04.2014.
URL: *http://davidlynch.org/blog/2011/09/why-not-just-use-an-ide-if-you-want-ide-features/*

McConnell, S. (2004), *Code Complete: A Practical Handbook of Software Construction*, 2nd edn, Microsoft Press, Redmond, WA, USA.

Moggridge, B. (2007), *Designing Interactions*, The MIT Press, Cambridge, MA, USA.

Raymond, E. S. (1999), *The Cathedral and the Bazaar*, O'Reilly Media, Sebastopol, CA, USA.

Raymond, E. S. (2003), *The Art of Unix Programming*, Addison-Wesley, Boston, MA, USA.

Simpson, K. (2014), *You Don't Know JS: Scope & Closures*, O'Reilly Media, Sebastopol, CA, USA.

Victor, B. (2012), 'Learnable Programming'. Accessed: 07.04.2014.
URL: *http://worrydream.com/#!/LearnableProgramming*

von Oldenburg, T. (2014), 'Analyzing JavaScript Scope'. Accessed: 12.05.2014.
URL: *http://www.protoandtype.com/analyzing-scope/*

List of Figures

Appendix

Sample source code

The following source code has been used for sketching during ideation and during prototyping.

```
;(function( window, document, $, undefined ) {
  console.log( this );
  $( document ).ready( function() {
    console.log( this );
    $('#btn').on('click', function( event ) {
      console.log( this );
      event.preventDefault();
      $.getJSON('document.json', function( data ) {
        console.log( this );
        data.forEach( function ( entry ) {
          console.log( this );
          $('#list').append('<li>' + entry.title + '</li>');
        });
      });
    });
  });
}( window, document, jQuery ));
```

Listing 8.1: Sample client-side JavaScript program

```
var fs = require('fs'),
    http = require('http'),
    marked = require('meta-marked'),
    handlebars = require('handlebars');

function parseMarkdown( cb ) {
  fs.readFile( 'article.md', function( err, data ) {
    if ( err ) {
      return cb( err );
    }

    var text = marked( data.toString() );
    cb( null, text );
  });
}

function compileHtml( text, cb ) {
  fs.readFile( 'template.html', function( err, data ) {
    if ( err ) {
      return cb( err );
    }

    var template = handlebars.compile( data.toString() );
    cb( null, template({
      meta: text.meta,
      content: text.html
    }) );
  });
}

var server = http.createServer( function( req, res ) {
  parseMarkdown( function( err, data ) {
    compileHtml( data, function( err, data ) {
      res.end( data );
    });
  } );
});

server.listen( 12345 );
```

Listing 8.2: Sample server-side JavaScript program