Visualising The Complex Features Of Source Code

Ivan Bacher [Thesis]
Technological University Dublin, ivan.bacher@tudublin.ie

Follow this and additional works at: https://arrow.tudublin.ie/sciendoc

Part of the Computer and Systems Architecture Commons

Recommended Citation

This Theses, Ph.D is brought to you for free and open access by the Science at ARROW@TU Dublin. It has been accepted for inclusion in Doctoral by an authorized administrator of ARROW@TU Dublin. For more information, please contact yvonne.desmond@tudublin.ie, arrow.admin@tudublin.ie, brian.widdis@tudublin.ie.

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 License
Visualising The Complex Features
Of Source Code

Author: Ivan Bacher
Supervisors: Prof. John D. Kelleher, Dr. Brian Mac Namee

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy
in the
Applied Intelligence Research Center
School of Computer Science

February 14, 2019
Declaration of Authorship

I, Ivan BACHER, declare that this thesis titled, “Visualising The Complex Features Of Source Code” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at the Technological University Dublin.

- Where I have consulted the published work of others, this is always clearly attributed and I have acknowledged all main sources of help.

- This thesis was prepared according to the regulations for postgraduate study by the Technological University Dublin and has not been submitted in whole or in part for an award in any other Institute or University.

- The work reported in this thesis conforms to the principles and requirements of the Technological University Dublin’s guidelines for ethics in research.

- Technological University Dublin has permission to keep, lend, or copy this thesis in whole or in part, on condition that any such use of the material of the thesis be duly acknowledged.

Signed: 

______________________________________________

Date: 

______________________________________________
“By visualizing information, we turn it into a landscape that you can explore with your eyes, a sort of information map. And when you’re lost in information, an information map is kind of useful.”

David McCandless
Abstract

Software development is a complex undertaking composed of several activities that include reading, writing, and modifying source code. Indeed, previous studies have shown that the majority of the effort invested in software development is dedicated to understanding code. This includes understanding the static structure, dynamic behaviour, and evolution of the code. Given these particular characteristics, as well as the high complexity of source code, it is reasonable to consider how visualisation can facilitate source code understanding.

This work proposes to extend existing software development tools with visualisations that can be used to encode the various complex features within a source code document. Further, this work establishes a design space which includes a series of visualisations that are meant to complement existing textual views of source code, as found in source code editors. Several prototype visualisations are presented in this work which utilise various visualisation techniques, such as tree visualisations techniques and visualisation techniques based on the code-map metaphor.

This work also presents several experiments, where the results of these experiments indicate that combining existing software development tools with visualisations can have a positive effect on source code understanding. Additionally, the result of the experiments show that the less a visualisation is abstracted from the original textural representation of a source code document, in terms of line, token, ordering, and character information, the more likely it is that viewers can link the visualisation to the code and back.
Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisors John Kelleher and Brian Mac Namee for their continuous support, patience, motivation, and immense knowledge. Their guidance helped me greatly throughout the Ph.D. program and especially during the writing of this thesis. I could not have imagined having better advisors and mentors. For that, I am truly thankful.

I am grateful to Alexandru Telea and John McAuley for accepting to be my examiners. I would like to thank TU Dublin for giving me the opportunity to partake in the Ph.D. programme and for providing funding via the Fiosraigh Scholarship Programme.

I would like to thank my family: My parents, Patrizia and Christoph, who gave me a fantastic start in life and raised me stubborn enough to survive the Ph.D. experience. I would like to thank Daniela and Luigi for supporting me throughout the last couple of years. I would like to thank my brother, Arno, who tested my prototypes, reviewed my surveys, and for all the insightful discussions.

I am sincerely grateful for all the support my extended family provided over the course of my Ph.D. studies. Special thanks are also directed to all my friends for the good times outside of work.

Last but not least, I would like to thank Victoria. Thank you for comforting me at the end of a long day. Thank you for challenging my ideas, reviewing my slides, and listening to my presentations. Thank you for your unconditional love. All this could have never happened without you.
# Contents

Declaration of Authorship iii

Abstract vii

Acknowledgements ix

1 Introduction 1

2 Background 11
   2.1 Software Visualisation .................................................. 11
      2.1.1 What is software visualisation? ................................... 11
      2.1.2 Improving Code Readability ....................................... 16
      2.1.3 The Code-Map Metaphor ............................................. 18
      2.1.4 Code Editor Integration ............................................. 24
      2.1.5 Evaluation Of Software Visualisations ............................ 26
   2.2 Source Code Understanding ............................................. 28
      2.2.1 Program Comprehension Strategies ................................. 29
         Terminology ............................................................ 29
         Bottom-up comprehension ............................................ 30
         Top-down comprehension ............................................. 30
         Systematic and as needed ............................................ 31
         Knowledge-based ..................................................... 31
         The Integrated Meta-model .......................................... 32
      2.2.2 Developer Information Needs ....................................... 32
      2.2.3 Source Code Understanding Issues ................................. 35
## 7 The Design Space Of Integrated Source Code Visualisation

### 7.1 Motivation

### 7.2 The Design Space

#### 7.2.1 Abstraction Characteristics
- Code layout
- Underlying Concepts And Constructs
- Ordering
- Characters/token information

#### 7.2.2 Abstraction Levels
- Abstraction Level 0
- Abstraction Level 1
- Abstraction Level 2
- Abstraction Level 3
- Abstraction Level 4

### 7.3 Summary

## 8 From Concrete to Abstract

### 8.1 Prototypes
- Prototype 1
- Prototype 2
- Prototype 3
- Prototype 4
- Interactions

### 8.2 Experiment

### 8.3 Results and Discussion

## 9 Conclusion

### 9.1 Summary

### 9.2 Limitations

### 9.3 Future Work
List of Figures

2.1 Branches of Visualisation .............................................. 12
2.2 Aspects Of Software ......................................................... 12
2.3 Code City ................................................................. 13
2.4 Tarantula ................................................................. 14
2.5 CVSscan ................................................................. 15
2.6 Formatted Source Code ..................................................... 16
2.7 Control Structure Diagram ............................................... 16
2.8 SeeSoft ................................................................. 19
2.9 Code-map Metaphor - Pixel Line 1 ...................................... 20
2.10 Code-map Metaphor - Pixel Line 2 ...................................... 20
2.11 Code-map Metaphor - 3D Pixel .......................................... 21
2.12 Code-map Metaphor - 2D Pixel .......................................... 21
2.13 Code-map Metaphor - Scaled Font ...................................... 21
2.14 Atom + Code Mini-map ..................................................... 23
2.15 Source Code Properties ................................................... 36
2.16 Example Of Scope Within Source Code ........................... 38

3.1 Icicle Tree ................................................................. 46
3.2 Circular Treemap .......................................................... 46
3.3 Experiment Web Application Screenshot ........................... 49
3.4 Experiment Framework ................................................... 50

4.1 HTML Code Fragment ...................................................... 56
4.2 Prototype Visualisation .................................................... 57
List of Tables

1.1 Contributions Corresponding To Chapters And Publications . 7
8.1 Dunns Test Results ............................................. 111
A.1 Technologies Used ............................................. 130
D.1 Prototype URLs ................................................. 140
Chapter 1

Introduction

Developing software is a complex undertaking, composed of several high level activities. These activities include designing, writing, modifying, and verifying the structure, behaviour, and evolution of source code. The Linux Information Project\(^1\) defines source code as “the version of software as it is originally written (e.g. typed into a computer) by a human in plain text (i.e. human readable alphanumeric characters), where the broader term software refers to all operating systems, application programs, and data that is used by products containing central processing units.” Previous studies have shown that software developers\(^2\) dedicate the majority of their time to understanding source code [15, 67]. Hence, a major part of the effort invested in software development is devoted to understanding the many kinds of structures, relationships, and hierarchies within the code [63, 69, 39]. Consequently, this makes source code a fundamental asset of the software industry.

Source code has several particular characteristics [70]. For one, entities within source code have many attributes that express their semantics (i.e. comments that decorate specific statements, and signatures of functions). Further, the scale of modern software systems can range from a couple of

\(^1\)http://www.linfo.org/source_code.html
\(^2\)We do not differentiate between software developers and programmers within this work, therefore we use the terms interchangeably
hundreds of lines of code to up to millions of lines of code. Finally, pro-
gramming languages used to write source code have strictly defined gram-
mars with non-ambiguous semantics. Given these particular characteristics,
as well as the high prominence and complexity of source code within the
software industry, it is reasonable to consider how visualisation can facilitate
source code understanding, as it is recognised that visual representations tap
into the capabilities of the powerful and highly parallel human visual sys-
tem [54]. Indeed, it has been shown that using visualisations to encode the
various aspects of source code can facilitate source code understanding. For
example, Hendrix et al. [31], Miara et al. [52], and Wettel et al. [75] illus-
trate the effectiveness of a software visualisation in regards to source code
understanding activities through controlled experiments.

Source code can be written in any of the hundreds of programming lan-
guages that have been developed to date, for example C, C++, C#, Java,
JavaScript, Python, Lua, and Perl. There are many different programs that
can be used for writing source code, ranging from simple general purpose
text editors, to source code editors, to fully fledged integrated development
environments. Source code editors are designed specifically for writing and
editing source code, which makes them a fundamental tool for software de-
developers. To facilitate source code understanding, it is important to maximise
the readability of source code [13]. Therefore, source code editors typically
have features such as syntax highlighting and pretty printing, which facil-
itate the process of viewing, reading, and modifying source code. This is
done by making the structural and syntactical composition of a source code
document more visible, through the use of indentation, spaces, line-breaks,
colour, and font-face.

Previous studies have shown that the typographic appearance of source
code [7] and the indentation style used within source code [52] can influ-
ence the speed and accuracy of program understanding. However, source
code documents are frequently too large to be displayed on a single screen. This introduces the need for scrolling, which can cause a cognitive burden for the user who must mentally assimilate the overall structure of the information space and their location within it [14]. Additionally, source code editors fall short in providing developers with information corresponding to the hierarchical structure, dynamic behaviour, or evolution of a source code document. Further, an open question remains on how to efficiently present programmers with various details within the code that can be examined in the text of the code (i.e. the mechanics of classes and methods) or concepts and structures that are not manifested directly within the code (i.e. concrete high level concepts such as modules and namespaces) [12].

In this work we propose to combine visualisations with existing software development tools (source code editors) to facilitate the process of understanding the various details embedded within a source code document. Therefore, we aim to design, implement, and evaluate a series of prototypes, where the goal of these prototypes is to facilitate source code understanding.

The main research questions we address are:

1. Do programmers make use of source code visualisations? This research question is answered in Chapters 4, 5, 6, and 8.

2. In which cases are source code visualisations that are integrated into source code editors helpful and in which cases are they unhelpful? This research question is answered in Chapters 4, 5, 6, and 8.

3. Are there visual representations that programmers find easier to link back to the text view of the code, and if there are, what are the characteristics of these visual representations? This question is answered in Chapters 7 and 8.

The main contributions of this thesis, by chapter, are:
Chapter 1. Introduction

1. An extensive analysis of the code-map metaphor and existing visualisations employing this metaphor, in terms of real world applicability, limitations, and perceived usability. The key finding being that software visualisation employing the code-map metaphor are widely perceived by the research community to be useful for software development due to the fact that the visualisations have a natural and direct mapping from the visual representation to the source code and back, resulting in high levels of trust on behalf of the user, which is supported by several qualitative observations. However, to date, little to no quantitative data exists in the literature that supports the claim that the use of the code-map metaphor can facilitate the process of software development. More information can be found in Chapter 2.

2. An investigation of the issues programmers face in regards to the many complex features of source code, specifically corresponding to scope, the scope chain hierarchy, and information related to the scope chain within source code. More information can be found in Chapter 2.

3. HTML structure visualisation prototype (Chapter 4)

   (a) The design and development of a prototype visualisation, which utilises a tree visualisation technique to provide viewers with an overview of the hierarchical structure of a source code document.

   (b) The evaluation of this prototype in the context facilitating of code understanding. The key finding being that an inappropriate visualisation can actually have a detrimental effect on task performance, while at the same time giving participants false confidence that they are performing well.

4. Code mini-map (Chapter 5)
(a) The design and development of a code mini-map visualisation, which utilises a visualisation technique based on the code-map metaphor and encodes additional details derived from a source code document.

(b) The evaluation of this visualisation in the context of facilitating code understanding. The key finding being that a code mini-map visualisation can successfully be augmented with additional encodings that represent the various complex details within a source code document, which in turn can lead to a positive effect on a programmer's code understanding.

5. Scoped (Chapter 6)

(a) The design and development of a novel interactive visualisation tool (Scoped) aimed to help programmers understand scope and the scope chain hierarchy within a source code document.

(b) The evaluation of this prototype tool in the context of facilitating code understanding. The key finding being that combining visualisations with a source code editor can have a positive effect on code understanding, especially in situations where the textual representation of the code no longer corresponds to the actual behaviour of the code.

6. The design space of integrated source code visualisation (Chapter 7 and Chapter 8)

(a) The definition of a design space for source code visualisation.

(b) An analyse and categorisation of the visualisation prototypes within the design space in terms of their level of abstraction from the textual representation of a source code document.
(c) The design and implementation of a series of prototype visualisations, each of which belongs to a different category within the design space, and all of which can be combined with existing software development tools to facilitate the process of reading, writing, and modifying source code.

(d) A controlled experiment evaluating of the implemented prototypes from the design space in the context of facilitating code understanding, with a key finding being that the more abstract a visualisation is perceived to be, the smaller the positive effect on code understanding.

The outlined contributions have been documented, in part, within the following publications:


Chapter 1. Introduction


As a summary, the contributions of this work, the corresponding chapters of this thesis, and the publications are shown in table 1.1. Further, urls to interactive versions of the prototypes that this thesis includes can be found in Appendix D.

**Table 1.1: Contributions corresponding to chapters and publications**

<table>
<thead>
<tr>
<th>Contribution</th>
<th>Chapter</th>
<th>Publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>[3]</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>[1]</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>[4]</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>[6]</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>[2, 1, 5]</td>
</tr>
<tr>
<td>6</td>
<td>7 &amp; 8</td>
<td></td>
</tr>
</tbody>
</table>

The remainder of this work is structured as follows. Chapter 2 presents a brief overview of software visualisation and includes various examples of software visualisations and details regarding evaluation in the context
of software visualisation. Furthermore, the chapter also presents information corresponding to source code understanding and program comprehension, including information on the issues developers face when working with source code, as well as the different cognitive models used by software developers.

Chapter 3 presents the methodology followed during the design and implementation of the presented prototype visualisations. Further the chapter also presents the methodology followed during the design, implementation, and execution of the presented experiments, as well as the processes used to analyse the results of the experiments.

Chapter 4 presents a software visualisation prototype in combination with a source code editor, which is used to represent the static structure of a source code document. Further, the chapter includes an experiment in the context of source code understanding, as well as an analysis and discussion of the experiment results.

Chapter 5 presents a mechanism for augmenting an existing software visualisations technique based on the code-map metaphor to encode additional information related to the hidden conceptual structures within a source code document. The chapter also includes an experiment related to source code understanding, as well as an analysis and discussion of the experiment results.

Chapter 6 presents a software visualisation prototype which combines an overview visualisation and a detail view visualisation with a source code editor. The prototype encodes a conceptual structure within a source code document and presents viewers with additional information corresponding to this conceptual structure. The chapter also includes a user evaluation study and a series of controlled experiments related to source code understanding, as well as an analysis and discussion of the study and experiment results.
Chapter 7 establishes a design space for integrated source code visualisation that can be used to describe a series of possible design choices for implementing visualisations designed to aid in the understanding of the various complex details within a source code document. The chapter includes details corresponding to the main motivations and goals of the proposed design space, as well as a series of prototypes and examples corresponding to the design space. Further, the chapter includes a series of metrics that can be used to categorise source code visualisations based on their level of abstraction from the textual representation of the code.

Chapter 8 presents four prototype software visualisations, all of which belong to the proposed design space of integrated source code visualisation (introduced in Chapter 7). Further, the chapter presents details of the experiments conducted using these prototypes to facilitate source code understanding, as well as an analysis and discussion of the experiment results.

Finally, Chapter 9 includes a brief summary of the presented work and discusses limitations and future work possibilities.
Chapter 2

Background

The following chapter includes a brief overview of software visualisation and presents various examples of software visualisations and details regarding the evaluation of software visualisations. The chapter also presents information about source code understanding, including issues software developers face and cognitive models used by software developers when working with source code.

2.1 Software Visualisation

As software systems and aspects related to these systems can be seen as a kind of abstract information, the following section present the use of visualisation in the context of software development to foster understanding and insight.

2.1.1 What is software visualisation?

Software visualisation (softvis) is a branch of information visualisation, depicted in Figure 2.1, and can be defined as the “the art and science of generating visual representations of the various aspects of software and its development process” [22, p.4]. These aspects include program code, requirement and design documentation, changes to source code, execution traces, and bug reports. The
goal of software visualisation is to help stakeholders to comprehend software and to improve productivity within all aspects of the software development life-cycle [12]. Researchers in software visualisation are concerned with visualising the structure, behaviour, and evolution of software (Figure 2.2).

Structure, refers to the static parts and relations of a system. For example, information that is valid for all possible executions of a software system [12, 39, 22]. This includes, program code, data structures, static call graphs, and the organisation of a program into modules. Behaviour, refers to the visualisation of the dynamic aspects of a software system, providing information about a particular run of the system [12, 39, 22]. Depending on the programming language, the behaviour of a system can be viewed on a higher level of abstraction such as functions calling other functions, or objects communicating with other objects. Finally, evolution refers to the development process of a software system in order to emphasise the fact the program code is changed over time to extend the functionality of the system or simply to remove bugs [12, 22].
Figure 2.3 illustrates an example of a software visualisation that is concerned with visualising the structure of a software system. *Code City*, developed by Wettel et al. [74, 73, 75], uses a city metaphor to depict software systems as three-dimensional cities, where classes are buildings and packages are districts. The figure shows the ArgoUML, an open-source Java project to draw and generate UML diagrams, code base as a 3D city, which is composed of 136000 lines of code, 14221 methods, 2522 classes/interfaces, and 1443 packages. Various software metrics can be mapped onto the visual properties of city artefacts. For example, the number of methods can be mapped to the height of the buildings, the number of attributes onto the base size, and the number of lines of code onto the colour of the buildings (from a dark grey that encodes a low line count, to an intense blue that encodes a high line count). The approach can also be extended to address the visualisation of design problems [73], where a certain colour is mapped to the affected artefacts according to the design problems that characterise them.

Figure 2.4 illustrates an example of visualising the behaviour of a software system. *Tarantula*, developed by Jones et al. [34], encodes the results of test
cases and maps these results back to the source code of a software system. Tarantula uses colour to show which lines of code contain faults. Red lines map to lines of code that have failed a test case and green lines map to lines of code that passed a test case. Yellow, is used to shown a combination of passing and failing test cases. Tarantula aims to help developers in locating faults in a program by illuminating possible faulty statements.

Figure 2.5 shows a screenshot of CVSscan [71], which is a software visualisation system that visualises the evolution of a source code file. CVSscan uses a single view to display the entire evolution of a code file. Each version of the file is drawn in a column using a visualisation based on the code-map metaphor [3], where every line of code is drawn as a pixel line on the screen. Successive versions of the file are shown in successive columns. Colour encodes line status. Green encodes constant, yellow modified, red modified by deletion, and light blue modified by insertion. Additionally, light grey
2.1. Software Visualisation

**Figure 2.5**: CVSscan: evolution of a single file. Figure from [71]

encodes inserted and deleted lines.

These examples have shown how visualisations can be used to encode the structure, behaviour, and evolution of source code. For the remainder of this work we will be focusing on visualisations that provide information about the structure of source code, specifically the structure of the code within a source code document. Therefore, visualisations that encode features within a code document which related to the behaviour and evolution of source code are out of scope for this work.
2.1.2 Improving Code Readability

To facilitate source code understanding, it is important to maximise the readability of source code [13], therefore, source code editors typically have features, such as syntax highlighting and pretty printing, which facilitate the process of viewing, reading, and comprehending source code. This is done by making the structural and syntactical composition of a source code more visible, through the use of indentation, spaces, line-breaks, colour, and fontface. Indeed, previous studies have shown that the typographic appearance of source code [7] and the indentation style used within source code [52] can influence the speed and accuracy of program comprehension. Figures 2.6 presents a formatted source code fragment and shows how white-space, more specifically indentation, can be used to improve code readability by making block structures more visible [13, 52].

Cross et al. [19, 31] go even further and introduce the control structure diagram (CSD), which is a graphical representation that maps directly to source code and augments the program text in order to make the nesting and control flow of the code more explicit [22]. Figure 2.7 illustrates how control structure diagrams (CSD) can be applied to formatted plain text source code. Cross et al. state that many programmers consider the source code to be the
only trusted specification of the software, therefore, the visualisation adds graphical constructs to the code, without disrupting its familiar appearance.

The CSD makes the control structures and control flow more visually apparent compared to the plain text alone (Figure 2.6). For example, vertical lines are drawn over the code to show the extent of code blocks, and vertically stretched oval lines are used to show the extent of loops. Furthermore, diamond glyphs are used to indicate the alternatives of conditional statements. The primary purpose of the CSD is to reduce the time required to comprehend source code by clearly depicting the control constructs and control flow at all relevant levels of abstraction. A controlled experiment was conducted to evaluate the effectiveness of CSDs [31]. The results of the experiment show that CSDs have a positive effect on program understanding in regards to shortening response times and increasing correctness of responses to questions posed about code [31]. This can be seen as initial evidence that integrating visualisations into existing software development tools can facilitate the process of understanding source code.

Although the CSD experiments have shown that integrating visualisations within a source code editor can help with source code comprehension, care must be taken in the design of these visualisations. There are a number of factors that need to be considered, including limited screen real estate, long-distance dependencies between structural elements of the code, and the requirement to enable a user to locate their current position within the ‘big picture’ of the source code. For example the available screen real estate of a source code editor is typically limited and source code documents are frequently too large to be displayed in the available space. This introduces the need for scrolling, which can cause a cognitive burden for the user who must mentally assimilate the overall structure of the information space created by a source code document and their location within it [14]. Further, Clifton
[13] states that the usefulness of the techniques used to improve the readabil-
ity of code diminishes when parts of the code or control structures within
the code are widely separated or heavily nested. Additionally, while source
code editors provide users with an overview of the package-class-method-
statement hierarchy in a software project, they typically fall short in provid-
ing the user with an overview of the hierarchical structure of a source code
document, which at the same time encodes the users current location within
the information space. Thus, exacerbating the cognitive burden introduced
by scrolling.

2.1.3 The Code-Map Metaphor

SeeSoft [23] is a well known example of a software visualisation. The original
publication has over 800 citations. SeeSoft was pioneered by Eick et al. in the
1990s to visualise the structure and evolution of large and complex software
systems. Figure 2.8 shows a screenshot of the SeeSoft system, visualising
several files containing over five thousand lines of code. Colour is used to
show code age, where red depicts recently modified code and blue depicts
code that has been unchanged for a long time. The smaller window shows
the source code corresponding to the current region in focus. Several facts
about the code base are shown including an overview of the relative sizes of
all files in the code base as well as their structure.

Marcus et al. [50] suggest that SeeSoft is so successful because it incorpo-
rates a natural and direct mapping from the visual metaphor to the source
code and back, leading to a natural navigation between the representations.
This makes the visual representation easy to understand, yielding high levels
of trust on the part of the user.

Many visualisations have been developed that are based on the SeeSoft
representation and several terms have been used to describe these types of
2.1. Software Visualisation

**Figure 2.8:** SeeSoft - Visualising the structure and evolution of a software system. Figure from [23]
visualisations, including dense layouts for text, pixel oriented views, and zoomed out views. However, none of the above terms are able to include all visualisations based on the SeeSoft representation, therefore, we will be using the term code-map metaphor. The code-map metaphor can be defined as “the mapping of source code to a zoomed out representation, either by the use of pixels, pixel lines, or a scaled down representation of text, in order to allow stakeholders to comprehend various statistics collected at the level of detail of individual lines of code” [6, p.1]. This term best describes the original intent of software visualisations based on the code-map metaphor as these encode source code to a zoomed out representation that maintains the spatial relationships between source code elements and visually encodes key metrics describing the characteristics of these source code elements.

There are three main approaches to mapping source code to a code-map representation. The first approach maps each line of code to a pixel line,
2.1. Software Visualisation

**Figure 2.11:** Pixel 3D. Figure from [49]

**Figure 2.12:** Pixel 2D. Figure from [49]

**Figure 2.13:** Scaled font. Figure from [21]
seen in Figures 2.9 and 2.10. Figure 2.9 uses the actual layout of the code, including indentation and spacing. Figure 2.10 ignores the layout of the code and maps each line of code to a new line in the visual representation. The second approach (Figures 2.12 and 2.11) maps each line of code to a pixel (2D) or cuboid (3D). The third approach (Figure 2.13) is similar to the first approach, however, it utilises a scaled down font representation instead of pixel lines. This approach also takes the original layout of the code into consideration. These three different forms of mapping from source code to visualisation will form part of the concrete to abstract source code visualisation design space that will be developed in Chapter 7 and explored in Chapter 8.

The most commonly used approach for mapping lines of code to a display medium is the pixel line approach depicted in Figure 2.9. Colour is an important attribute and is used in most tools as a means for encoding additional information. For example colour can be used to encode the results of queries, which lines of code contain faults, and code age. The literature indicates that software visualisations based on the code-map metaphor are perceived by the research community to be helpful across all aspects of the software development process. However, to date this hypothesis has not been rigorously tested empirically.

Source code editors are common tools that programmers use when writing, reading, or modifying code. All of these tasks involve understanding the microscopic and macroscopic details within the code [12]. The microscopic details include the mechanics of classes and methods which can be examined in the text of the code, while the macroscopic details include concrete high level concepts such as modules, system, and conceptual structures that are not manifested directly in the code. Many editors, including Atom\(^3\) and Sublime Text\(^4\), incorporate a visualisation which presents the viewer with a

\(^3\)https://atom.io/
\(^4\)https://www.sublimetext.com/
2.1. Software Visualisation

The code mini-map visualisation is based on the code-map metaphor and acts as an overview component mapping source code to a zoomed out representation, either by the use of pixels, pixel lines, or a scaled down representation of text, presenting viewers with a zoomed out view of the currently open source code document showing the layout of the code. This is important due to the fact that the layout of the code, as intended by the original developer, often conveys a great deal of information (e.g. associations or relations by physical proximity) [48]. Figure 2.14 shows an example of a code mini-map being used in a code editor.

In summary, a source code editor provides programmers with a view of the textual representation of the code and a code mini-map provides programmers with an overview of the code. The code mini-map is based on the code-map metaphor, which in turn is also employed by several well known software visualisations tools such as SeeSoft [23] and SV3D [49]. Section 2.1.2
presents three main factors that need to be considered when designing code visualisations. These are limited screen real estate, long-distance dependencies between structural elements of the code, and the requirement to enable a user to locate their current position within the ‘big picture’ of the source code. We believe that the code mini-map visualisations succeeds in all three of these factors. First, by presenting viewers with a zoomed out view of the currently open source code document, the code mini-map overcomes the factor of limited screen real estate. Second, by using the original layout of the code (including indentation) the code mini-map explicitly shows the nesting structure of the structural elements within the code document. Finally, by highlighting the current line on which the cursor is located within the code editor, the code mini-map enables users to locate their position within the code document. However, an open question remains regarding how to add additional encodings to the code mini-map, where these encodings correspond either to the microscopic details (i.e. mechanics of classes and methods) or the macroscopic details (i.e. conceptual structures that are not manifested directly in the code) within the code.

2.1.4 Code Editor Integration

A subset of source code editors include a type of visualisation similar to the code mini-map presented in Section 2.1.3. However, many source code editors do not include any types of code visualisations. These source code editors fall short in providing users with an overview of either the structure, behaviour, or evolution of the code, while at the same time encoding the users current location within the information space. Therefore, due to the fact that source code editors are the most commonly used tools for working with source code [12], we believe that integrating visualisations into to these tools can bring many positive benefits to software developers.
Cherubini et al. [12] conducted a study which investigated how and why software developers use visual representations when working with code. The results of the study show that the main reasons why developers use visual representations are to better understand, design, and communicate the various aspects related to source code. Developers typically maintain complex mental models of their code during development and use visual representations to update and expand these models as familiar code evolves over time and as a means of exploring unfamiliar code [12]. Cherubini et al. [12] suggest that software developers might benefit from an interactive visualisation, that allows for the exploration of low level details while remaining in the high level structure, as no current view conveys both levels of this abstraction simultaneously.

Cox et al. [17] conducted an experiment to investigate the utility of variable dependency diagrams. Participant comments and feedback from the experiment provides important insights into the characteristics of effective software visualisations. Further, a list of suggestions for software visualisation designers, based on the comments and feedback of participants, is provided. The most important ones being:

1. Different visualisations should be available for different tasks, all of which support both top-down and bottom-up program comprehension. The visualisations should provide developers with different views of a software system. Each of the visualisations should only concentrate on presenting viewers with a small subset of information, in an obvious and clear manner, as many developers will only use the visualisations when the needed information is obvious and easily obtained. Miller’s findings in relation to how many pieces of information a typical human can remember (7 plus/minus 2) should be taken into consideration when designing a visualisation [53].
2. Information must relate to the source code. Programmers are very code-centric when working with code, therefore, visualisations need to clearly link visual representations or encodings back to the code. Independent, stand alone tools that do not link information back to the code are frequently left “alone tools”. Visualisations divorced from the source code need to clearly link back to the code.

3. Information must be integrated as programmers want and need information to be provided in the context of their working environment. Visualisations are only useful when they satisfy the current information needs of programmers.

Following these findings and suggestions, we believe that integrating a variety of visualisations, each of which explicitly focus on a different feature of source code that developers find difficult to understand, can facilitate the software development process.

2.1.5 Evaluation Of Software Visualisations

While many software visualisation approaches have been proposed during the last decade, relatively few have been empirically validated so far [40]. Hence, there is a growing need for objective assessments of visualisation approaches to demonstrate their effectiveness [75]. One of the reasons behind the shortage of empirical evaluations lies in the difficulty of performing a controlled experiment for software visualisation, as organising and conducting a proper controlled experiment is a difficult endeavour. Wettel et al. [75] conducted a survey of research works dealing with experimental validation of software engineering, information visualisation, and software visualisation approaches. The survey presents several guidelines on how to design and conduct controlled experiments successfully. The guidelines are as follows [75]:
1. Choose a fair baseline for comparison.

2. Involve participants from industry.

3. Take into account the range of experience level of participants

4. Provide a tutorial of the experimental tool to the participants.

5. Find a set of relevant tasks.

6. Include tasks which may not advantage the tool being evaluated

7. Limit the time allowed for solving each task

8. Choose real-world systems

9. Include more than one subject system in the experiment design

10. Provide the same data to all participants

11. Report results on individual tasks

12. Provide all the details needed to make the experiment replicable

To evaluate the effectiveness of a software visualisation during a controlled experiment, researchers have investigated several different ways to measure the comprehension of a software system. These include think-aloud protocols and the use of comprehension tasks [63].

For think-aloud protocols, participants verbalise their thoughts. These are audio or video recorded. The recordings are then transcribed, and the transcriptions are used to analyse the thought process of the participants. For comprehension tasks, participants are given a set of tasks to complete, where the tasks require participants to understand the structure, behaviour, or evolution of a software system. For each task, the time it takes to complete the task and task correctness is measured. These measurements are then analysed and can be compared to results for similar tasks.
The drawback of using think-aloud protocols is the effort it takes to complete such a protocol, as the data needs to be recorded, transcribed, and then analysed. As a result, many researchers avoid this effort. The drawback of using comprehension tasks is that it is difficult to come up with a set of reliable tasks, as the tasks should be based on real word examples. Additionally, participants of all skill levels should be able to complete the tasks. Another problem with comprehension tasks lies in the fact that comparative studies often report overall performance for a combined set of tasks. However, Plaisant [60] suggests that reporting results per task is preferable. Many of the comprehension tasks that were created for previous studies also seem unusual and inappropriate in regards to the development challenges today’s developers face [63].

We will return to the design of software visualisation evaluation experiments in Chapter 3 where we present our experimental methodology, and also in Chapters 4, 5, 6 and 8 where we present experiments evaluating different software visualisations.

### 2.2 Source Code Understanding

It is difficult to consider software visualisation without also considering the task or tasks it is meant to support. Further, it is unlikely that any single software visualisation can simultaneously address all issues related to source code understanding. Hence, to design, develop, and implement an interactive software visualisation that facilitates source code understanding, we must first examine the cognitive models of program comprehension strategies, the information needs of software developers, as well as the comprehension difficulties associated with source code.
2.2. Source Code Understanding

2.2.1 Program Comprehension Strategies

Many studies have been conducted to observe how programmers understand programs [68]. Due to this, several cognitive models of program comprehension strategies have been proposed which attempt to describe the behaviour of these programmers. However, before we review these cognitive models we must first define some terminology.

Terminology

A *mental model* describes a programmers mental representation of the program to be understood, as opposed to a *cognitive model*, which describes the cognitive processes and temporary information structures in the programmers head [69]. These cognitive processes and temporary information structures in turn are used to form the mental model [69]. A mental model, in the context of software development, is an internal working representation of a program. The model contains entities such as text structures, chunks, plans, and beacons [51].

*Text-structure* knowledge includes the program text and its structure [51]. This type of knowledge is stored in a programmers long term memory and grows through experience. Examples of text-structure knowledge units are control structures, loop constructs, variable definitions, module calling hierarchies.

*Chunks* are knowledge structures containing various levels of text-structure abstractions [51]. Chunks can be split into microstructures and macrostructures. Macrostructures correspond to a programs text control-flow organisation and only include labels, while microstructures include all statements. Lower level chunks can form higher level chunks and higher level chunks comprise several labels and the control flow relationships between them.
Plans are knowledge elements for developing and validating expectations, interpretations, and inferences. They capture the comprehender’s attention during program understanding tasks. Plans can also include casual knowledge about the information flow and relationships between parts of a program.

Beacons are recognisable, familiar features that act as cues and typically indicate the occurrence of certain structures within the code [8]. Brooks [8] states that there can be multiple beacons for a given structure or operation, and that different beacons may indicate the presence of these structures with different probability. Further, the same feature may participate in beacons for a variety of structures or operations.

Bottom-up comprehension

Shneiderman [62] proposed that programs are understood in a bottom-up fashion. For example, when programmers read source code, they mentally chunk low-level software artefacts into meaningful high-level abstractions. These high-level abstractions are then further grouped until a high-level understanding of the program is formed. Pennington [56] reports observations of programmers using the bottom-up strategy during the gathering of statement and control-flow information.

Top-down comprehension

Brooks [8] suggests that programs are understood in a top-down manner. This means that programmers restructure knowledge about the application domain and map this knowledge to the source code. The top-down strategy starts with a global hypothesis about the program. The initial hypothesis is then refined into a hierarchy of secondary hypotheses, which are further refined and evaluated in a depth first manner. Verifying or rejecting a hypothesis depends on the presence or absence of beacons [8]. Researchers [66] have
2.2. Source Code Understanding

observed this comprehension strategy being used when the code or type of code is familiar.

Systematic and as needed

In a study where programmers were given the task of enhancing a personnel database program, Littmann et al. observed two comprehension strategies [47]. One comprehension strategy was that programmers systematically read the code in detail, tracing through the control and data-flow abstractions in the code to obtain a global understanding of the program. Programmers using this strategy acquired both information about the structure of the program and interactions between components in the program when it executed. This enabled them to form a mental model of the program. The other comprehension strategy was that the programmers focused only on the code relating to a particular task, i.e. as needed approach. Programmers using this strategy, however, only acquired information about the structure of the program. Thus, forming a weaker mental model compared to programmer that used the systematic approach.

Knowledge-based

Letovsky [45] suggests that programmers are capable of exploiting either the bottom-up or top-down comprehension strategies. The theory has three main components consisting of a knowledge base that encodes a programmer’s previous programming experience, a mental model that represents the programmer’s current understanding of the program, and an assimilation process that describes how the programmers mental model evolves using the programmer’s knowledge-base and program information. The assimilation process relies on inquiry episodes, which consist of a programmer asking a question, conjecturing an answers, and then searching through the code and documentation to verify the conjecture.
Chapter 2. Background

The Integrated Meta-model

Mayrhauser and Vans [51] developed the integrated metamodel, which combines the top-down, bottom-up and knowledge based approaches into a single model. Moreover, Mayrhauser and Vans propose that program understanding is built concurrently at several levels of abstraction. This is done by being able to freely switch between the three comprehension strategies. Observations from experiments show that programmers frequently switch between all three comprehension strategies.

2.2.2 Developer Information Needs

A major part of a software developers work is answering questions about code [36, 64, 43, 44]. Therefore, by better understanding these questions, as well as by extracting the challenges developers face when attempting to answer these questions, we can gain insight into the information needs of software developers during the process of working with source code.

Letovsky [45] reports that, at the most general level, software developers ask “why” (purpose of action or design), “how” (the way some goal of the program is accomplished), “what” (meaning of variables), “whether” (if code behaves in a certain way), and “discrepancy” (observations do not match expectations) questions. It is important to note that the questions were not classified simply on the basis of specific words used, rather, on the basis of what the questions are asking about.

Ko et al. [36] observed 17 software developers at a large company and were able to extract details about the information the developers sought, the sources they used, and data corresponding to the situations that prevented important information from being acquired. Using these observations Ko et al. were able to create a list of 21 questions which correspond to interactions with code, other software artefacts, and teammates. Further, Ko et al. report
that developers ask design questions about the code rational and implications of a change. Important findings of the study include that when writing code, developers seek functionality to reuse and information on how to reuse it. Developers submitting a change to a code base ask if the change is correct, whether it follows team conventions, and what changes it should include. When receiving a new bug report, developers first try to reproduce the bug to determine what it looked like and when it occurs before asking about its cause.

Sillito et al. [64] conducted two studies, one carried out in a laboratory setting and the other in an industrial setting, which observed programmers working on a range of change tasks using a series of different programming tools. The goal of these studies was to collect information to better understand what programmers need to know about a code base when performing a change task, how a programmer goes about finding information, and how well current programming tools help during the process of performing a change task. Sillito et al. were able to identify 44 questions that software developers asked about code. Some of the main findings include, at the beginning of a coding task developers attempt to find focus points that correspond to domain concepts or application functionality and work outwards following relationships between methods and classes. Also, developers ask higher-level questions about relationships between multiple methods and classes, including questions about control and data flow.

Fritz and Murphy [26] interviewed eleven experienced software developers and present 78 questions that software developers ask, which reveal the range of interpretations developers desire for compositions of software project information. The questions span across multiple domains that include source code, change sets, teams, work items, web sites and wiki pages, comments, exception stack traces, and test cases.
LaToza and Myers [44] surveyed 179 professional developers about hard-to-answer questions they asked during the process of creating, debugging, and understanding code, in order to get a better understanding of a developer’s information needs. The survey synthesises 94 distinct questions which could be split into 21 categories. The five most frequent categories are rational (Why wasn’t this done in another way?), intent and implementation (What does this do?), debugging (How did this error occur?), refactoring (How can I refactor this without breaking existing use cases), and history (Who, when, how, and why was this code changed or inserted?). Many of the questions that LaToza and Myers were able extract were highly focused around specific hypotheses, situations, or proposals relevant to a developers current task and mental model of the code. Further, LaToza and Myers suggest that a key design goal for tools or languages should be to better use the situations described in developers’ questions to focus and filter the information the tools and languages provide to developers.

What all of these studies have found is that many of the questions software developers ask are highly focused and correspond to a particular feature or aspect that is relevant to a developers current task and mental model of the code. Using this information, we believe that it is important for a software visualisation to focus on providing visual answers to these specific questions. Further, a single visualisation should not attempt to answer every question a developer might ask. These findings are reflected in the remainder of this thesis, where we present a series of design guidelines and visualisation prototypes which attempt to facilitate the process of code understanding by explicitly focusing on a single complex aspect within a source code document.
2.2.3 Source Code Understanding Issues

People have been developing software for over 40 years and although there have been major advances in terms of programming languages, programming constructs, source code editors, and integrated development environments the process of writing software can still be described as difficult. There are many reasons that this process is difficult including that computers still require very clear instructions and the human brain is not wired to efficiently think in terms of these instructions. Further, to produce any meaningful piece of software, developers need to typically be efficient in several programming languages. For example, when creating a web application, a programmer should at least be proficient in HTML, CSS, and JS. To know any one programming language reasonably well typically requires learning and remembering hundreds of different facts consisting of operations, concepts, and design patterns specific to that language.

During a coding session, developers are required to keep several things in their working memory at the same time. This means that when writing or modifying a piece of code the corresponding developer has to create a mental model of how that specific code fragment functions, including variables used, dependencies, and other relationships and hierarchies. For example, during the process of creating a moderately complex class the issues that need to be kept in mind include how the class will get created, how it will be used, and how it will use other classes. When working with existing code, a developer might have to be thinking through several different layers of looping, conditionals, return statements, and branches.

Source code has several distinctive properties. Figure 2.15 illustrates a subset of these properties and comprehension difficulties associated with each of the properties. Source code is written in programming languages that have strictly defined grammars with non-ambiguous semantics [70]. It
can be composed of up to millions of lines of code [70]. Source code contains many kinds of relations such as the types of variables, members and parents of classes, dependencies of packages, and interfaces of modules [70]. Further, source code contains many types of hierarchies, such as the package-file-class-method-statement hierarchy, scope chain hierarchies, control structures hierarchies, and hierarchies of data structures [70]. Additionally, entities within source code have many attributes that express their semantics, such as access rights for interface members, comments that decorate specific statements, and signatures of functions [70].

As previously noted in Section 2.1.3, Cherubini et al. [12] state that programmers need to understand both the microscopic and macroscopic details within the code, where the microscopic details include the mechanics of classes and methods which can be examined in the text of the code and
the macroscopic details include concrete high level concepts such as module, system, and conceptual structures that are not manifested directly in the code.

An example of a macroscopic detail within source code is the scope chain hierarchy. The scope chain hierarchy is a fundamental conceptual structure implemented by almost every programming language, and can be seen as a set of rules that control the visibility and lifetime of variables, functions, and parameters [18, 65]. This hierarchy is important to the programmer because it can be used to reduce identifier naming collisions and also provides automatic memory management [18]. Two aspects of the scope chain hierarchy can be a cause of confusion for programmers. First, each programming language has a slightly different implementation of scope. Second, because the scope chain can be seen as a type of hierarchy, it supports nesting. Therefore, scopes can be nested within each other, meaning that if an identifier (variable, function, or parameter) cannot be found in the immediate scope, the corresponding scope chain is traversed, starting at the parent scope-level and continuing until the identifier is found, or until the outermost (global) scope has been reached. Figure 2.16 illustrates an example of scope and the scope chain within a source code fragment. The example contains three nested scopes, where scope 1 encompasses the global scope and has one identifier in it: foo. Scope 2 encompasses the scope of foo, which includes the three identifiers: a, bar, and b. Finally, scope 3 encompasses the scope of bar, and it includes one identifier: c.

Fard and Mesbah [24] propose a list of code smells for applications written in JavaScript. Code smells are patterns in source code that indicate potential comprehension and maintenance issues. Once detected, code smells need to be re-factored to improve the design and overall quality of the code [24]. The list is composed of 13 code smells, where 7 are existing well-known
smells adapted to JavaScript, and 6 are specific JavaScript code smells, collected from various JavaScript development resources. The smells related to scoping issues include: `this` keyword usage, variable naming conflicts, scope chaining, nested callbacks, and excessive global variables. Interesting questions that developers ask in the context of scope issues include: Where (in which scope) was a specific variable defined and where in the code can a global variable be changed.

Due to the importance of scope and scope chain hierarchy in this thesis we will use the scope chain as a case study to explore the use of source code visualisations to improve source code understanding. In particular, Chapters 5, 6, and 8 will present experiments that evaluate visualisation of the scope chain. In preparation for this work, we extended the work of Fard and Mesbah [24] in terms of the information requirements of programmers, specifically ones related to scoping issues. To gather these information requirements an analysis of the 50 most popular stack overflow questions was conducted. The question retrieved were tagged with the keywords “scope” and “JavaScript” and were categorised into 4 broad categories: Identifiers
(36%), this context (36%), design issues (17%), and other (17%). The identifiers category deals with issues regarding the declaration, accessibility, and state of an identifier (variable, function, and parameter). The this context category is concerned with issues relating to the this keyword associated with each individual scope, as the use of the keyword within source code has specific behaviour associated with it. The design issues category includes issues corresponding to design decisions programmers make during the process of writing and modifying code. Finally, the other category includes the usage of specific methods and frameworks. Many maintenance and comprehension issues are due to the poor understanding of scope. Hence software developers struggle with answering questions such as: in which scope is an identifier declared, can an identifier be accessed from the current scope, and is an identifier that has already been defined within the parent or global scope being overwritten.

2.3 Summary

This chapter began with a brief introduction to software visualisation, highlighting that researchers within the field of software visualisation are concerned with visualising the structure, behaviour, and evolution of software. A series of examples was shown that illustrated how visualisations can be used to encode these three aspects of software. Further, we stated that for the remainder of this work, our main focus was on visualisations that provide software developers with information about the structure of source code. More specifically, the structure of the code within a source code document.

Next we showed how visualisations can be used to improve code readability, as this is an important aspect in the context of facilitating code understanding. We presented three main factors that should be considered during the process of designing these visualisations. The three factors are limited
screen real estate, long-distance dependencies between structural elements of the code, and the requirement to enable a user to locate their current position within the ‘big picture’ of the source code document.

A series of source code visualisations that are based on the code-map metaphor are presented. These visualisations are perceived to be successful by the research community because they incorporate a natural and direct mapping from the visual metaphor to the source code and back, leading to a natural navigation between the text and visual representations. A specific visualisation based on the code-map metaphor was introduced, which can be described as the code mini-map.

The code mini-map visualisation can be integrated into existing source code editors and acts as an overview component, mapping source code to a zoomed out representation to present viewers with a ‘birds eye’ view of the currently open source code document while still showing the original layout of the code. We illustrated how the code mini-map can overcome the three factors that should be considered during the process of designing a visualisation to facilitate code readability and code understanding. Further, we state that an open question remains of how to incorporate additional encodings to the code mini-map, which correspond either to the microscopic details or the macroscopic details within a source code document.

Several cognitive models of program comprehension strategies were presented that attempt to describe the behaviour of programmers during the process of working with source code. Further, the information needs of software developers were included, which emphasise the fact that a major part of a software developers work can be seen as answering questions about code. Therefore, visualisations that encode the complex features of source code should be tailored toward answering the questions that developers ask when working with code. A series of code understanding issues are presented, as well as an analysis of the question that developers ask when facing
these issues. An important concept related to code understanding issues can be described as scope and the scope chain hierarchy. Hence, we use issues related to the scope chain as a case study to explore the use of source code visualisations to improve code understanding. In Chapter 3 we present the main design methodology followed during the development of the visualisation prototypes presented in Chapters 4, 5, 6, and 7. Further, we also include the experimental methodology used in the experiments which evaluate the newly created visualisation prototypes.
Chapter 3

Methodology

The previous chapter presented a brief overview of software visualisation and several different software visualisation systems were shown. Further, information regarding the evaluation of software visualisations was presented and various issues software developers face when dealing with source code were also described. This chapter presents the design and experiment methodology used in the remainder of this thesis. The design methodology entails the rational followed during the process of designing and implementing the software visualisation prototypes presented in this work. The experiment methodology includes a high level description of the experiment framework used for conducted controlled experiments that evaluate the effectiveness of adding visualisations to source code editors, with the goal of facilitating source code understanding.

3.1 Design Methodology

To help information and software visualisation designers, practitioners, and researchers, several design guidelines have been proposed [72, 57, 58, 61, 17, 22, 32]. While many of these guidelines are applicable, they do not necessarily correspond particularly well to creating visualisations that encode the complexities within a single source code document, as well as integrating these visualisations into existing software development tools (source code
Chapter 3. Methodology

editors), where the main goal is to facilitate source code understanding. However, these guidelines were used as inspiration from which we were able to synthesise a subset that correspond particularly well to the process of creating integrated source code visualisations. These guidelines can be described as follows: language specific, order adjacency, awareness, target users, and natural mapping.

3.1.1 Language specific

The goal of source code visualisation is to provide the viewer (a software developer) with additional information that corresponds to a specific aspect within the code. However, to date, there are many different types of computer languages, including Java, C#, JavaScript, C++, F#, HTML, XML, CSS, and Python. Each of these computer languages contain their own set of semantics, relations, hierarchies, and rules. For example, HTML imposes a very strict document structure, in terms of the code constructs used. Therefore, a visualisation which display an overview of this document structure could be very useful to a programmer when working with a large HTML source code document. However, the same visualisation might not be as useful when encoding the document structure of a computer language which does not impose a strict document structure, such as CSS or JavaScript. Further, Java contains a Package-File-Class-Method-Statement hierarchy, while Python is more flexible and does not impose this strict hierarchy. Hence, a visualisation that concentrates on encoding the strict hierarchical structure within a code document is probably of more use to a programmer when working with Java compared to a programming language such as Python.

Another example includes the implementation of scope rules. These differ depending on the programming language, and so, some programming languages are more prone to cause scope understanding issues than others
3.1. Design Methodology

[39]. JavaScript can be considered one of these languages, as the syntax of the language makes false promises that can lead to errors [18, 65]. Further, the textual representation of the code (in JavaScript) does not always correspond to the actual behaviour of the code, therefore, many understanding issues can arise from this specific limitation. Hence, a visualisation which encodes the complex scope rules or hierarchies within a JavaScript source code document may be of more use compared to the same visualisation which encodes the scope rules of a source code document written in C#.

We believe that a visualisation must be tailored towards a specific programming language and that the effectiveness of the visualisation also changes depending on the programming language.

3.1.2 Order Adjacency

In many programming languages, the ordering of code elements is particularly important as it specifies the control and execution flow of the code. Therefore, it may be advantageous to preserve the ordering of code elements in any visualisation so as to avoid confusing the viewer. For example, when visualising the code elements within a HTML document, an icicle tree diagram [30] (Figure 3.1) provides a projection of the ordering that also aligns with the indentation of the code, as opposed to a circular tree-map (Figure 3.2) where the ordering of each node is calculated by a circle packing layout algorithm. This distinction between visualisations that maintain order adjacency and those that do not is one of the factors included in the visualisation design space developed in Chapter 7, and evaluated in Chapter 8.
3.1.3 Awareness

Software developers become code-centric when working with source code, therefore a visual representation of source code needs to clearly link the encoded information back to the textual representation of code, especially when tasks involve source code manipulation [17]. Further, software developers want and need information to be provided in the context of their working environment, which is a source code editor. Cordy [16] states that to achieve success, tools must present information in the context of source code and provide strong coupling between code and the visualisations. This means that as soon as changes are made to the code, the visualisations must reflect these changes.

3.1.4 Target users

The main target users of source code visualisations can be defined as anyone working with source code. It is important to note, that the main aspect that differentiates these target users, is their programming experience level. This is a complex aspect, as a programmer can be an expert in one programming
language, and a novice in another programming language. Further, a programmer can also be an expert or a novice in terms of their general programming experience, which includes high level programming concepts which do not necessarily link to a specific programming language.

3.1.5 Natural mapping

During the process of designing and implementing a source code visualisation, it is important to consider the aspect within the code to be visualised and how this aspect maps to the visual representation. For example, scope or the scope chain hierarchy within a source code document can be described as a series of neatly nested “bubbles” that each act as a container, in which variables, functions, and parameters are declared [65]. A natural mapping from the concept of scope to the visual representation could be the use of a visualisation technique that shows parent child relations using containment, as this emulates the “bubbles” metaphor previously used to explain the concept of scope. Similarly, when showing hierarchies, tree visualisation techniques are well suited for this task (e.g. circular treemaps (Figure 3.2), icicle tree diagrams Figure 3.1).

Referring back to Section 2.1.3, it may also be beneficial to choose a visualisation metaphor that depicts the original layout and structure of the code, in terms of the textual representation of the code. Visualisations which employ the code-map metaphor incorporate a natural and direct mapping from the visual representation to the source code and back. This can lead to more natural navigation between the representations, which makes the visualisation easier to understand and yields a high level of trust on behalf of the user.
3.2 Experimental Methodology

Chapters 4, 5, 6, and 8 of this thesis describe experiments designed to measure the effect of different visualisations on a software developer’s understanding of the hierarchies and relationships that exist within a source code document. This section describes the general methodology used to design and execute these experiments, as well as the processes used to analyse the data generated from them.

Conducting controlled experiments is an important part of software visualisation research. These experiments provide quantitative and qualitative data on the usability and usefulness of the prototypes. Further, the experiments also provide more information on the visualisations effect on aspects such as code understanding for proof-of-concept prototypes to state-of-the-art software visualisation tools.

A difficulty of conducting experiments and/or evaluations is that it is time consuming and difficult to recruit participants. Also there is a high a risk of inconclusive results if not enough participants are recruited [37, 10]. To overcome this difficulty, each of the experiments conducted for this research project were conducted as remote experiments. Bruun et al. [9] state that designing a study which can be performed remotely can not only lower the barriers to participation, but also make it possible to obtain larger sample sizes that in turn can increase statistical power when analysing results. For example, Ko and Wobbrock [38] were able to recruit hundreds of developers in just one hour, avoiding the logistics of scheduling a lab-based, in-person study. Further, Ko et al. [37] suggest that remote participation, depending on the study, can also allow participants to use their own workstations, their own space, their own tools, and even work at times of their choosing, potentially making the experiment more ecologically valid by accounting for participants’ natural work environments.
3.2. Experimental Methodology

Figure 3.3: Experiment web application screenshot

All experiments described in this thesis were implemented as web applications. Figure 3.3 shows a screen shot of the experiment web application used to evaluate the prototype presented in Chapter 5. The figure shows a question/task being presented to the viewer, a source code document, and a source code visualisation. A demo version of this experiment can be found here: https://ivanbacher.github.io/p-016-experiment-testing. The choice of creating web applications was based on the criteria that the experiments should be supported by as many different systems as possible. We believe that this approach minimises the cost for people to “give it a try” [20], therefore increasing the chance of large numbers of participants. More details about the technologies used to create the experiments can be found in Appendix A. Further, we have created a list of important things to consider when creating web based experiments and we have also included a list of suggested web platforms on which one can recruit participants.
3.2.1 Experiment Process Overview

Figure 3.4 illustrates the process that was used to implement and conduct the experiments presented in Chapter 4, Chapter 5, and Chapter 6, and Chapter 8. The process consists of seven high level steps and can be used for the rapid design of future experiments. More information corresponding to these steps is presented below.

**Step 1:** Participants are presented with general information describing the experiment. This information states the goal of the experiment, as well
as the main motivation for the experiment. This step also involves presenting participants with a consent form. The form includes information corresponding to the participant’s rights, the data captured during the experiment, how the captured data will be analysed, and where the captured data is stored. Before a participant can continue to the next step, a checkbox had to be checked to verify that the participant has accepted the consent form. The consent form used is included in Appendix B.

**Step 2:** Participants are presented with several questions all of which ask to self estimate their general programming experience and language specific programming experience. This is an important point as a participants programming experience could greatly influence the outcome of an experiment.

**Step 3:** Participants are given a series of definitions which are related to the terminology used during the experiment. For example, when dealing with the scope chain hierarchy, participants are given the definitions of scope, parent/child scope, etc.

**Step 4:** Participants are given a chance to get familiar with the prototype used within the experiment. The step usually involves presenting the participant with a walkthrough of the prototypes main functionality, as well as explaining all encodings present in the current prototype visualisations. Further, the participant is usually given a series of “dummy” questions/tasks to make the learning process seem more natural.

**Step 5:** This step presents the participant with a series of code understanding questions or tasks, and can be seen as the main part of the experiment.

**Step 6:** Participants are presented with a questionnaire which collects information about their subjective opinion of the visualisation prototype
used for the current experiment. Participant are also given a chance to add any comments or other feedback.

**Step 7:** The last step of the experiment process presented participants with a chance to share the experiment on social media sites. This falls in line with a sampling technique called snowball sampling [28], where existing study subjects play a part in recruiting future subjects, through their social networks.

### 3.2.2 Participant Recruitment, Data, and Procedure

Ko et al. [36] state that recruiting can be seen as a marketing problem, where the problem is to get the attention of a set of people with certain characteristics. Hence, there are two goals. The first on being to find out where potential participants put their attention and the second goal being to figure out how to entice potential participants to participate [36].

One group of potential participants for our remote experiments is students. Invitation emails were sent out to computer science students across the different universities. Further, invitation pamphlets were also distributed during computer science lectures. To entice these students to participate, an opportunity to win a 15 Euro/Dollar Amazon.com or Amazon.co.uk voucher was offered once the experiment was completed.

Software professionals were also targeted, however, in contrast to students, these professionals can be difficult to recruit (mainly due to the fact that they have busy schedules [36]). Email invitations were sent out to companies and invitations to participate were posted on social media platforms such as reddit and twitter. These participants were also offered an opportunity to win an amazon voucher once the experiment was completed.

During the experiments, various data about each participant was captured. This data consisted of programming languages expertise, as well as
subjective opinions and ratings corresponding to the prototype systems. Further, demographic data related to a participants age and gender were also collected. After that, participants were directed to one of the experiment treatments groups (all of the experiments followed a between subject design where multiple treatment groups were present).

Once the walkthrough and tutorials were completed, participants were presented with the experiment tasks or questions, which were presented one by one. Generally, the participants were allotted a specific time limit for each task/question, as per guidelines from Wettel et al. [75] on conducting controlled experiments in the context of software visualisation. After answering all of the presented tasks/questions, participants were asked to complete a short exit questionnaire that collected various subjective ratings or demographic data.
Chapter 4

Visualising The Structure Of

HTML

A drawback of working with source code editors is that the display medium used to display the code is typically limited in size and the code documents are frequently larger (in terms of lines of code) than the available display space. This introduces the need for scrolling, which can cause a cognitive burden for the user [14]. Therefore, we believe that visualisations which presents viewers with an overview of the structure of the current code document are a possible solution to the problem of limited display space. This chapter describes a visualisation prototype that presents the viewer with an overview of the structure of HTML source code documents and an experiment that evaluates the usefulness of this visualisation prototype.

Figure 4.1 shows an example of a source code fragment written in HTML (Hypertext Markup Language). HTML is composed of a series of elements, which are the building blocks of webpages. HTML provides a means to create structured documents by denoting structural semantics for text, such as paragraphs, headings, lists, quotes, and other items. Web browsers do not display the HTML code, but use it to interpret the content of a webpage.

Figure 4.1 shows that HTML imposes a strict hierarchical structure, as HTML elements are nested within each other. Furthermore, HTML documents are typically larger than the available display space, in terms of lines of
code. Therefore, practitioners working with HTML could profit from a compact visualisation, that is combined into an existing source code editor, which encodes the hierarchical nesting structure of an HTML code document.

### 4.1 Prototype

Figure 4.2 shows a screenshot of the prototype interface designed to include an overview visualisation of a HTML document, which is composed of an icicle tree diagram and a source code editor. Nodes within the icicle tree diagram encode the hierarchical structure of the HTML document, located in the source code editor, and are represented as graphic primitives. Parent-child relationships are encoded by horizontal adjacency, meaning that child nodes are placed to the right of their parent. Depending on which structural element the text cursor is located at within the source code editor, the corresponding node within the icicle tree is highlighted. The layout of the icicle tree diagram is calculated using a space-filling algorithm, in order to use

![HTML code fragment](image-url)
the available display space and avoid scrolling. However, this introduces a limitation that at a certain number of nodes, the visualisation becomes too packed to comprehend. We designed our experiments to avoid this scenario.

### 4.2 Experiment

The experiment followed a between group design and was implemented using the online experimental platform described in Section 3.2. Two versions of the application were hosted online, one that included the overview visualisation and one that omitted it. 39 subjects participated in the experiment, of which 25 were currently enrolled in a computer science, web development, or software engineering course. 17 participants completed the experiment using the prototype with the visualisation, while 22 completed the experiment using the prototype without the visualisation.

As described in Section 3.2.1, the experiment consisted of a series of steps and took approximately 30 minutes to complete. First, participants were presented with a consent form and general information about the experiment.
Next, participants were presented with definitions for terms such as node, child node and ancestor node in order to become accustomed to the wording used within the experiment. Thereafter, participants were shown an image of the prototype, which included several annotations. The annotations were used to describe functionality such as a text search feature, or to describe the visualisation. Participants were presented with three source code documents and for each document they were asked to answer a set of questions. The questions as well as the source code documents were given to the participants in random order. Question accuracy and response time were gathered. After the main experiment, participants were asked to complete an outgoing questionnaire. Participants that completed the experiment with the overview visualisation were presented with two additional questions in the questionnaire. These required them to rate the usefulness of the overview visualisation.

Three HTML source code documents of different complexity were used in this experiment. We measure complexity using lines of code, total nodes, and nesting levels. The easy source code document consisted of 47 lines of code, 8 nesting levels, and 30 nodes. The medium source code fragment consisted of 247 lines of code, 6 nesting levels, and 167 nodes. The hard source code fragment consisted of 2,787 lines of code, 13 nesting levels, and 1,605 nodes. Participants were asked to answer a series of questions for each source code document. The questions can be categorised into two types depending on the task the participant had to perform, counting or locating. Examples of the questions are shown below, where “X” and “Y” were replaced with unique identifiers that identified nodes within the source code using the HTML id attribute, e.g. id = “ZBHRB”. For a full list of questions we refer the reader to Appendix C.

- How many child/descendant/leaf nodes does node “X” contain?
4.3 Results and Discussion

In the experiment 39 participants completed 819 questions and accuracy as well as response time was recorded for each question. Figures 4.3 and 4.4 depict the percentages of correct answers for counting and locating tasks corresponding to each source code fragment.

![Figure 4.3: Counting task accuracy (A - without overview visualisation, B - with overview visualisation)](image)

- What node is the closest common ancestor of nodes “X”?

In regards to participant accuracy, an interesting pattern emerged when comparing questions based on task type. For questions that involved counting nodes, accuracy increased when the overview visualisation was present (Figure 4.3) for all three source code documents. This was surprising, as we only expected participants to have a higher accuracy score in the medium source code document due to the fact that the easy source code document only consisted of 30 nodes, and the hard source code document tested the limitations of the space-filling layout algorithm used for the overview visualisation. We suggest that this increase in accuracy is due to the fact that the visualisation fundamentally encodes structural elements of source code, thus making them easier to count.
For questions which involved locating nodes, accuracy decreased when the overview visualisation was present (Figure 4.4). This could be due to many factors, however, we believe that the main reason for the observed decrease in accuracy during the locating tasks was that the overview visualisation only explicitly encoded the current node in which the cursor was located, within the source code editor. Therefore, by adding additional encodings to the overview visualisation, such as the use of colour for depicting previously visited nodes, we believe that participant accuracy can be improved.

In terms of question completion times, participants were generally faster using the overview visualisation, however, this also led to a decrease in accuracy in certain questions. This is an interesting finding, as it suggests that using an inappropriate visualisation can actually have a detrimental effect on task performance, while at the same time giving participants false confidence that they are performing well. These results can be used as a first step towards illustrating the benefits and shortcomings of the use of tree visualisations in the context of providing an overview of the hierarchical structure of a source code document.
Chapter 5

The Code Mini-Map

The code mini-map shows the viewer an overview of a source code document that is being viewed in a source code editor. The code mini-map visualisation technique is used in many existing source code editors. While the visualisation technique is generally understood to be useful, limited experimental data currently exists on the effect the technique has on code understanding. A detailed review of the use of code mini-map visualisations is given in Section 2.1.3.

In this chapter, we describe a visualisation prototype based on the code mini-map metaphor, that extends the typical static view of code with a layered approach that allows extra information to be encoded. Further, we present an experiment which evaluates the effect of the visualisation prototype on source code understanding.

5.1 Prototype

Our code mini-map based prototype augments the code mini-map visualisation presented in Section 2.1.3 with a layering mechanism, which can be described as a way for providing additional information to the viewer, while using the same underlying visualisation (the code mini-map). This approach is similar to the way a user is able to add additional information to an online map, for example traffic layers, street name layers, or elevation layers.
The additional layers of information, can also be described as additional layers of encodings. These encodings can be used to represent either the microscopic details, the macroscopic details, or a combination of both within a code document. In our prototype the additional layer of information corresponds to scope and the scope chain hierarchy within a source code document. More information related to scope and the scope chain can be found in Section 2.2.

Figure 5.1 is a screenshot of the prototype, including a code editor combined with a code mini-map which includes the scope information layer. The yellow highlighted area on the code mini-map encodes the extent of the scope in which the cursor is currently located. Each time the cursor changes location within the code editor the code mini-map is updated. If a programmer is interested in seeing to which scope a specific variable belongs, then the programmer must highlight that specific variable within the code editor. Figure
5.2. Experiment

5.1 shows an example of this behaviour. In the example, the variable `value` is highlighted within the code editor. The red border on the code mini-map now shows the scope to which the variable `value` belongs. By looking at the code mini-map we can see that the variable `value` does not belong to the scope in which the cursor is currently located. Additionally, by placing the mouse pointer either in the yellow highlighted area or within the red border, the corresponding lines of code are highlighted within the code editor. Thus, the viewer can see which lines of code belong to either the scope in which the cursor is located or the scope to which a specific variable belongs to. An interactive demo of the prototype can be found here: http://tiny.cc/cmm-scope.

5.2 Experiment

The purpose of the experiment presented in this section is to evaluate the effectiveness of using the code mini-map to encode additional information that corresponds to a complex aspect within a source code document. In this case the complex aspect relates to the scope chain and information related to the scope chain within a source code document. The experimental methodology described in Section 3.2 was followed for this experiment.

The main research question we aim to address is: Does adding a scope information layer to a code mini-map visualisation increase the ability of programmers to understand the impact of scope on the correctness of their code?

To measure the effectiveness of adding additional layers of information to a code mini-map visualisation, 60 participants were randomly split into two groups where Group A was presented with a standard code mini-map visualisation and Group B was presented with a code mini-map visualisation that included additional encodings (the scope information layer). All participants were shown the same source code\(^5\) and given a set of questions that

\(^5\)http://tiny.cc/cmm-scope-data
corresponded to scope understanding issues. The questions given to the participants were used as a proxy to measure the effect of both versions of the code mini-map on source code understanding (if a question is answered correctly). All questions correspond to the following format: The variable “X” (line 3) belongs to ____________, where participants were tasked to fill in the name of the missing scope. “X” was replaced with the names of variable derived from the source code document shown to participants. For a full list of questions we refer the reader to Appendix C.

The programming language used in this experiment is JavaScript. According to the stack-overflow developer survey results 2016\(^6\), 2017\(^7\), and 2018\(^8\) JavaScript is currently the most popular programming language amongst StackOverflow users. The main factor that could influence a participant’s performance during the experiment is their JavaScript programming experience level. To control for this confounding factor, each participant was randomly placed into one of two groups (Group A or Group B). Additionally, each participant was also asked to fill out a short questionnaire at the beginning of the experiment and the results of this questionnaire were used in a post-hoc analysis of the experiment to ensure that the randomisation process worked correctly. The goal of this questionnaire was to gather information corresponding to each participant’s programming experience level. Participants were asked to enter their self estimated experience level using a 5 point Likert scale where each number within the scale was replaced with a level of Dreyfus’ model of skill acquisition [11, p.162]: 1) Novice, 2) Advanced beginner, 3) Competent, 4) Proficient, and 5) Expert.

Figure 5.2 shows the participants’ self estimated experience level with the JavaScript programming language for both groups of participants in the experiment. The images show that the both groups have a similar distribution

\(^6\)https://insights.stackoverflow.com/survey/2016
\(^7\)https://insights.stackoverflow.com/survey/2017
\(^8\)https://insights.stackoverflow.com/survey/2018
5.3 Results and discussion

All participants answered 20 code understanding questions related to scope understanding issues. Group A (code mini-map) consisted of 30 participants and Group B (code mini-map + scope info layer) consisted of 30 participants. In order to measure potential differences between the groups on an aggregated level, the number of correct answers given by each participant was counted.

Participants in Group A answered a total of 600 questions, where 356 (59.3%) were answered correctly and 244 (40.7%) incorrectly (Figure 5.3). Participants in Group B also answered a total of 600 questions, where 421 (71.2%) were answered correctly and 244 (29.8%) incorrectly (Figure 5.3). These results indicate that augmenting the visualisation with an overlay (as was done
for Group B) improved the accuracy of the respondents on this task. A Pearson’s Chi-squared test [59] shows that there is a statistically significant difference between the groups when comparing the percent of correctly answered questions (p-value <= 0.05).

Wettel et al. [75] suggest also reporting results on individual tasks/questions as this allows for a more precise and in-depth analysis of the strengths and weaknesses of an approach. Hence, to get a more precise and in-depth understanding of the number of correctly answered questions per group, results corresponding to each individual question were also analysed. Specifically, the number of correct answers for each individual question. Figure 5.4 displays a grouped bar chart, where each group of bars corresponds to an individual question and the length of each bar indicates the percentage of correct answers for that question. The colour of each bar represents to which group that bar corresponds.

By examining Figure 5.4, we notice two interesting things. The first is that for most of the questions (16/20), participants from Group B answered a greater number of questions correctly compared to participants from Group A. This result reinforces the earlier result indicating that augmenting the visualisation with an overlay (the Group B condition) can be beneficial. The

Figure 5.3: Percent of correct answers for all questions
second is that for 6 questions (Q3, Q5, Q11, Q12, Q14, Q15) the percent of correct answers is below 50% for both groups. This is interesting, as upon closer inspection, these questions were the only questions which corresponded to a specific, and tricky, feature of scope within JavaScript. This feature can be described as variable hoisting, e.g. variables declared in a block scope (if, while, for, and switch statements) are hoisted to the upper function scope rather than belonging to the block scope. Hence, the textual representation of the code no longer corresponds to the actual behaviour of the code.

These results confirm that a specific feature of a programming language, such as variable hoisting, can be difficult to understand for programmers. However, it is interesting to note that in 5 out of 6 of these difficult tasks (Q3, Q5, Q11, Q12, and Q14) Group B had higher accuracy. This result indicates that although these questions remained difficult even with the help of an augmented visualisation (correct answers remained below 50%) augmenting the visualisation does appear to have a positive impact. Therefore, we believe that visualisations can be particularly useful when the textual representation of the code no longer corresponds to the actual behaviour of the code (as is
the case, for example, in languages such as JavaScript that implement variable hoisting) and that these visualisations in turn, can be used to facilitate code understanding.

In summary, the findings from this experiment indicate that adding an additional layer of information to the code mini-map visualisation can have a positive effect on code understanding. We have shown that this is the case for scope understanding issues in the context of JavaScript source code. However, we also believe that these results can be generalised in terms of using the code mini-map visualisation to encode other microscopic or macroscopic details within a source code document.
Chapter 6

Scoped

In this chapter, we present a software visualisation prototype composed of two visualisations in combination with a source code editor. We call this prototype Scoped and an interactive demo of the prototype can be found at: http://tiny.cc/jsscope. The prototype encodes the scope chain hierarchy within a source code document and also presents the viewer with additional information that corresponds to the scope chain hierarchy, such as variables, function, and parameter declarations. Further, a user evaluation study and two experiments in the context of source code understanding are presented, as well as an analysis and discussion of the evaluation study and experiment results.

6.1 Prototype

Figure 6.1 shows a screenshot of Scoped. The tool consists of three main components: A) An overview visualisation encoding the scope chain within source code; B) The identifier (variable, function, and parameter) information panel; C) A source code editor. Colour is used to link each of the components. Within the overview visualisation, green is used to show in which scope the cursor is currently located within the source code editor; blue is used to represent the parent and ancestor scopes which the currently selected scope can access; and grey is used to represented the remaining scopes that
cannot be accessed. Additionally, the type of scope (function or block) is encoded using solid or dashed lines. The identifier information panel presents variables, functions, and parameters that can be accessed in the currently selected scope. Colour is used to show local identifiers (green) and identifiers belonging to parent or ancestor scopes (blue). The globe icon represents identifiers that belong to the global scope. Within the source code editor the currently selected scope is highlighted in green (as shown in the Figure 6.1)

Scoped has been designed with a number of common programming use cases in mind. For example, a common scenario in software development is that a programmer is tasked with re-factoring existing code. Before reading the code, the programmer can use Scoped to get an insight into the underlying complexity of a code fragment in terms of scope nesting. While reading code, a programmer typically has to scroll through it as the available screen real estate of a their display is limited. This can cause disorientation. The overview component of Scoped, however, can be used to limit disorientation by visually showing developers their current location within the scope chain hierarchy of a source code document. Furthermore, the overview component shows the scope in which the cursor is currently located, as well as
which scopes are accessible and the list view shows which identifiers can be accessed from the current cursor position within the source code document. This provides the programmer with a visual way of answering common questions related to the understanding of scope during the process of reading, writing, or refactoring code.

### 6.2 Experiment

A user experience evaluation and two controlled experiments were conducted using the Scoped tool. The user experience evaluation study was conducted to probe for further design requirements and to understand user behaviour when interacting with Scoped. The controlled experiments were conducted to evaluate the effectiveness of adding a composite visualisation to a source code editor, and to measure the effectiveness of each of the individual components within the composite visualisation in regards to facilitating the process of understanding scope relationships within source code. In this context, composite visualisations can be defined as a visualisations that consists of
two ore more individual components, as is the case for the Scoped prototype tool. The main research question we aim to answer is: *Does combining a composite visualisation with a source code editor facilitate the process of understanding source code?*

**Evaluation Study**

Five participants took part in the user experience evaluation study [42], which lasted for approximately 45 minutes for each participant. The majority of the participants were postgraduate students and none used JavaScript on a day to day basis. In regards to using students, Kitchenham et al. [35] state that using students as subjects is not a major issue as long as you are interested in evaluating the use of a technique by non-expert users.

The user evaluation study followed a think aloud protocol [55], where participants were encouraged to verbalise their thoughts as they moved through the user interface to complete the given tasks. Each participant was first given a short introduction to the visualisation tool, where the main features and functionality of the tool were explained. Next, each participant worked through a 15 minute tutorial together with the observer. After the tutorial, the participants were given 2 tasks to complete. The first task involved answering a series of questions corresponding to issues related to scope. The questions included asking which identifiers (variables and functions) belonged to a specific scope, moving the cursor to a specific line in the code and answering in which scope the cursor is currently located, as well as moving the cursor to a specific line in the code and answering whether a certain identifier was accessible from that line. The second tasks involved finding a bug within a source code fragment, where the bug was composed of a local variable using the same name as a variable defined within the parent scope. After the participants completed both of the tasks, they were asked to fill out a short survey, which included questions relating to their experience with the tool.
6.2. Experiment

We will discuss the results of the user evaluation study in the results section later in the chapter.

6.2.1 Controlled Experiments

The first controlled experiment evaluated the effectiveness of adding a composite visualisation to a source code editor and the second experiment evaluated the effectiveness of each of the individual components within the composite visualisation. Both experiments used a between-subject design and followed the experimental methodology described in Section 3.2. In the first experiment one group of participants (Group A) was presented with a standard source code editor with no visualisations present and the other group of participants (Group B) was presented with a source code editor that included the composite visualisation presented in Figure 6.2 (Figure 6.2 illustrates the scope labelling feature enabled, while this feature is disabled in Figure 6.1).

In the second experiment one group of participants (Group C) was presented with a source code editor that included only the overview component (component A from Figure 6.2) and the other group of participants (Group D) was presented with a source code editor that included only the detail view component (component B from Figure 6.2).

All participants (from both experiments) were presented with the same source code documents, which were obtained from public code repositories\(^9\) and the programming language used in the experiment was JavaScript. In both experiments participants were presented with source code documents and asked to answer a series of questions designed to interrogate their understanding of the scope chains within those documents. The questions were designed using an analysis of popular stack overflow questions related to understanding scope in source code [1]. Participants from both experiments were presented with the same set of questions. The ability of participants

\(^9\)Experiment data: https://www.tiny.cc/scoped-exp-data
to correctly answer questions was used as a proxy measure for their understanding of scope within source code documents. By measuring the differences in the ability of participants to correctly answer questions under different conditions we explore the effectiveness of the visualisations included in Scoped.

The main confounding factor that could influence a participant’s performance during the experiment is their JavaScript programming experience level. In order to control for this factor, participants were asked to enter their self estimated programming experience level [25] using a 5 point Likert scale, where 1 indicated very inexperienced and 5 indicated very experienced. Figure 6.3 shows the distribution of the participants’ self estimated experience level with the JavaScript programming language. The figure shows that all four groups have a similar distribution of novice to expert programmers. The results of a Kruskal-Wallis [41] test show that there is no statistically significant difference in JavaScript experience between the four groups (p-value = 0.846).
6.3 Results and Discussion

The following section presents a series of observations, which were gathered during the user experience evaluation. After discussing the observations from the user study we then present the results of the controlled experiment.

6.3.1 User Evaluation

The user experience evaluation study had two tasks, one in which users answered questions about scope in a source code document, and one in which they tried to debug an error. All of the participants frequently interacted with the visualisations within Scoped when attempting to answer the questions during the first task. Recall that the first task required the participants to answer a number of questions corresponding to issues related to scope. When answering the question of which identifiers belong to the global scope, none of the participants investigated the source code. Instead they moved the cursor to the global scope within the source code editor or clicked on the circle representing the global scope within the scope chain visualisation and then investigated the identifier information panel in order to answer the question. A participant commented “This is easier than looking through the code, all I have to do is click on a circle or move the cursor to a function and then I can find the information I am looking for within the list view”. When asked to which scope a specific variable belonged, the majority of the participants used the visualisation first, and then looked at the code. An interesting observation was that the participants seemed to use the visualisations to verify their assumptions before answering the questions.

During the second task (debugging) the participants mainly focused on the code in order to understand its behaviour. After the participants understood how most of the code worked, they attempted to look at the code in order to find any lines that might seem suspicious. When a suspicious line
was found, the participants glanced at the visualisations within the tool in order to investigate which identifier belonged to which scope and if an identifier could be accessed from the currently selected scope.

The participants brought several issues to our attention. For example, one participant stated that the identifier list view should support changing lines within the source code editor (e.g. navigating to a specific line of code where a specific variable is declared or accessed). While another participant stated that the link between the source code editor and the identifier information panel could be improved by adding a mechanism to highlight which variable is currently of interest to the user. Most participants also stated that it would have been helpful if labels were present within the scope chain visualisation.

We believe that the feedback and suggestions brought forward by the participants of the evaluation study revealed an important aspect of software visualisation: a source code visualisation should be linked/grounded to the code. This also reinforces our previous statements that relating the visualisation to the source code is important for programmers. This observation regarding the grounding of visualisation relative to the code contributed to the design space that is developed in Chapter 7.

The feedback and suggestions provided by the participants were integrated into the next design iteration of the Scoped tool, which was then used in two controlled experiments to evaluate the effect of a composite visualisation on code understanding. Figure 6.1 illustrates the version of Scoped that was presented to the participants of the user evaluation study, while Figure 6.2 presents the next version of Scoped. One of the main changes being that the new version of scoped includes scope labels, which can be enabled or disabled on the overview component. Further changes include performance, interaction, and rendering improvements.
6.3.2 Controlled Experiments

Two controlled experiments were conducted. The first controlled experiment evaluated the effectiveness of adding a composite visualisation to a source code editor and the second experiment evaluated the effectiveness of each of the individual components within the composite visualisation.

For the first experiment 88 participants were recruited. Each of these participants was randomly assigned to one of two conditions. Participants in Group A were presented with a source code editor with no visualisation (this was the baseline group), participants in Group B were presented with a source-code editor with the composite visualisation. 46 participants were allocated to Group A and 42 to Group B. All participants answered 6 code understanding questions related to the concept of scope. To measure potential differences between the groups on an aggregated level, the number of correct answers given by each participant was counted. Figure 6.4 presents the distribution of participant correctness scores for Group A and Group B. The plots show that the groups have different distributions and that a greater number of participants in Group B answered all of the questions correctly compared to Group A. Participants from Group A had an average correctness score of 4 and participants from Group B had an average correctness score of 5. A Kruskal-Wallis test shows that there is a statistically significant difference between both groups (p-value = 0.045). We believe that this is an interesting finding as it shows that participants presented with the composite visualisation were able to answer the code understanding questions with higher accuracy compared to participants that did not have access to the visualisation.

For the second experiment, an additional 57 participants were recruited, where 28 participants were allocated to Group C (overview visualisation) and 29 participants were allocated to Group D (detail view visualisation).
The participants were given the same code documents and code understanding questions as in the first experiment. Figure 6.4 presents the distribution of participant correctness scores for Group C and Group D. The plots show that correctness scores for Group C and D were quite similar. Participants from Group C and Group D had an average correctness score of 5. A possible reason for these results is that both the overview component and the detail view component, in terms of their level of abstraction from the original source code document, can be described as more abstract compared to a visualisation based on the code-map metaphor (Chapter 5). Therefore, participants may have found it difficult to link the encoded information presented within the individual components to the source code document located in the source code editor. More information relating to a source code visualisations abstraction level will be discussed in Chapter 7 and evaluated in 8.

When comparing the results of both experiments, participants in Group A (no visualisation) did worst and participants in Group B (composite visualisation) did best. A series of Kruskal-Wallis tests show that there is no statistical difference between any of the groups, except between Group A and Group B. Hence, the full composite visualisation is necessary for statistically significant positive effect on code understanding compared with the baseline.

To get a more precise and in-depth understanding of participant correctness scores, results corresponding to each individual question were also analysed. Figure 6.5 displays a grouped bar chart, where each group of bars corresponds to an individual question and the length of each bar indicates the percentage of correct answers for that question. By examining Figure 6.5, we notice that for most of the questions, participants in Group B (composite visualisation), Group C (overview component), and Group D (detail view component) answered a greater number of questions correctly compared to Group A (no visualisation). Further, it seems that for three out of the six
6.3. Results and Discussion

**Figure 6.4**: Distribution of participant correctness scores between groups in the two studies.

**Figure 6.5**: Combined results for both experiments showing the percent of correct answers for each question.
questions (Q2, Q3, and Q6) the percent of correct answers differs by a large amount for Group B (composite visualisation). After examining the source code associated with these questions more closely, we noticed that it incorporated a specific feature of JavaScript. The source code used for Q2, Q3, Q6 relied on hoisting [65, p.41], i.e. variables declared in a block scope (if, while, for, and switch statements) are hoisted to the upper function scope. This can be a great source of confusion for novice and experienced programmers alike, especially for programmers that are mainly familiar with languages such as C and Java that do not implement hoisting. We observed similar behaviour related to variable hoisting during a different experiment, which can be found in Section 5.3.

In summary, the results of the experiment described in this chapter show that combining a composite visualisation with a source code editor can have a positive effect on code understanding, especially in situations where the textual representation of the code no longer corresponds to the actual behaviour of the code. Additionally, the results show that both the overview and detail view components of the composite visualisation are important.
Chapter 7

The Design Space Of Integrated Source Code Visualisation

In the previous chapters (Chapter 4, Chapter 5, and Chapter 6) several prototype visualisations that integrated with source code editors and encoded different features of source code were presented. During the process of creating and evaluating these visualisation prototypes, a question arose of which type of visual representation was better suited for encoding the complex aspects within a source code document. Further, the presented prototypes can be distinguished by the amount that they abstract away from the textual representation of the underlying information space, which in this case corresponds to the textual representation of a source code document. For example, the prototype presented in Chapter 5 (the code mini-map) is less abstract than the prototype presented in Chapter 6 (Scoped).

To explore this more the following chapter establishes a design space for integrated source code visualisation, which describes a series of possible design choices when implementing visualisations that encode the various complex features within source code. The design space focuses on including visualisations that are meant to complement existing textual views of source code, as found in source code editors or integrated development environments. Hence, we use the term integrated source code visualisation to describe the design space. Source code visualisations that act as standalone tools are
not the focus of this work, therefore, they are not included in the proposed
design space.

Despite not being a space in the strict mathematical sense of a vector
space, a design space serves as an established analogy in many fields [61].
The goal of the design space is to drive the exploration of various different
types of visualisations that are created to represent source code, as well as
encode the many properties and characteristics of the code. The main goal
of integrated source code visualisations present within the design space is
to provide visual solutions to any code understanding issues a programmer
might face during the process of working with source code. The source code
visualisations belonging to the design space can be categorised, based on
their level of abstraction from the code.

In the following sections we first provide details corresponding to the
main motivations and goals of the proposed design space of integrated source
code visualisation. Then, the design space is described, including the cate-
gories to which visualisations can be added. Finally, various visualisation
prototypes and examples are presented.

7.1 Motivation

Source code is a complex phenomenon and there are many issues related to
understanding the structure, behaviour, and evolution of code. Source code
editors show the code in its written form, which can be seen as the lowest
level of abstraction when working with code. Hence, in a way programmers
are bound to the written form of their code. Seeing code in its pure text rep-
resentation is the standard way of working with code to date, however, this
approach still has some limitations. For one, text is not able to successfully
convey many of the hidden constructs, concepts, and structures within the
code.
This brings up the question of how to represent these various features within a source code document. Further, an additional question is how similar the visual representation should be to the underlying data. In this case the underlying data can be seen as the source code itself and as previously stated, developers are bound to the written form of their code. Hence, to which level of abstraction can a visualisation still be linked back to the code?

In order to attempt to answer these questions, we start by establishing a design space for integrated source code visualisation. An important note is that the visualisations belonging to the design space are not meant to be standalone tools, but rather complement existing software development tools or views (as the ones found in source code editors). This approach of combining visualisations into existing software development tools to provide viewers with two or more different views of a system can be explained with the help of an example from the video game industry. Figure 7.1 shows a screenshot from the Battlefield video game series. This is a first-person shooter, where the main view is from the viewpoint of the player’s character. To the bottom left of the Figure, we can see a type of mini-map that shows additional information to the player, such as allied units, enemies, objectives, and surroundings. The goal of this mini-map is to aid the player in orienting themselves within the game world.

Now linking this example to software development and software visualisation, the main view can be described as the textual view of source code, and the mini-map which provides additional information to the programmer can be described as the visualisations proposed in the design space. Hence, the main goal of the design space is to provide a framework for the exploration of various different types of visualisations that could be used to represent source code, as well as encode the many properties of the code.
Chapter 7. The Design Space Of Integrated Source Code Visualisation

7.2 The Design Space

The design space of integrated source code visualisation can be seen as a virtual space for collecting a variety of possible visualisation prototypes, all of which are meant to complement existing software development tools, to facilitate the process of working with source code. Figure 7.2 presents a screenshot of a source code editor, and shows where within the editor the visualisation prototypes belonging to the design space can be inserted. The inserted visualisation prototypes should focus on encoding the various features within the source code that is currently loaded in the source code editor. These aspects can be microscopic details, macroscopic details or a combination of both, derived for the source code document currently loaded the the code editor. The microscopic details include the mechanics of classes and methods which can be examined in the text of the code, while the macroscopic details include concrete high level concepts such as modules, system, and conceptual structures that are not manifested directly in the code.

A very important aspect of these types of split views is that the inserted
7.2. The Design Space

Visualisations need to be highly coupled with the text view of the code. This means that as soon as the code changes, the cursor changes, or the user interacts with the code, the visualisations need to update to represent the new state of the code. Additionally, the inserted visualisations should also support user interactions. This means that when the user interacts with the visualisations, the text view of the code should be updated accordingly. An example of this would be using the visualisations to scroll through an open code document or using the visualisation to jump to a section of the code which is of interest for a programmer when working on a specific task.

Figure 7.3 shows a subset of visualisation prototypes, all of which belong to the proposed design space of integrated source code visualisation. The visualisations in Figure 7.3 encode a hidden construct within a source code document (excluding visualisations A and B within Figure 7.3). This hidden construct can be described as the scope chain hierarchy (more information about this hidden construct can be found in Chapter 2). Further, the colour red (red border) is used to encode in which scope construct a variable of interest is currently located and the colour yellow is used to encode in which

```javascript
// Convert timestamp (milliseconds) into a time string in the format "(h)MM:ss".
function toString(timestamp){
    var HOUR = 3600000,
        MINUTE = 60000,
        SECOND = 1000;

    // Format timestamp into a time string.
    function calc(){
        var h = timestamp % HOUR,
            m = timestamp % MINUTE + 1,
            s = timestamp % SECOND + 1,
            str = "";

        str += h + ':' + 'm' + ':' + 's';
        str += pad(m, 2) + 's';
        str += pad(s, 3);

        return str;
    }

    return calc();
}

// Function to convert timestamp into a time string.
function countSteps(val, step, overflow){
    val = Math.floor(val / step);
}
```
Figure 7.3: Subset of prototypes within the design space of integrated source code visualisation
7.2. The Design Space

**Figure 7.4:** Design space abstraction categorisation
scope construct the cursor within the source code editor is currently located. It is important to note that the visualisations can be tailored to encode other information related to the code.

In this section and the following subsections, the design space of integrated source code visualisation will mainly be discussed along the lines of the prototypes presented in Figure 7.3.

7.2.1 Abstraction Characteristics

We have extracted several metrics that can be used to measure the level of abstraction of a visualisation from the code loaded within the code editor. These aspects include the layout of the code, underlying constructs within the code, ordering of the elements within the code, and the number of characters or tokens a specific statement is composed of within the code.

Code layout

The layout of the code, including whitespace and indentation, is a very important aspect of a source code document as it often conveys a great deal of information [48]. Deline et al. [21] present initial evidence that programmers form spacial memory of the layout of the code. Therefore, the closer a visualisation can mimic the original layout of the code, the stronger the coupling between the visualisation and the textual view of the code.

Underlying Concepts And Constructs

On one level, software developers need to examine the text of the code to get an understanding of the mechanics of classes and methods, as well as the flow of control structures [12]. On the other hand, software developers also need to grasp the complex higher-level concepts and constructs, such as modules, namespaces, and scope chain hierarchies [12]. These concepts
and constructs are typically not manifested directly within the code. Hence, the process of understanding these high level concepts and constructs often takes a great amount of time and can bring up several code understanding issues.

Efficiently showing details manifested directly within the code and details not manifested directly within the code is a challenging task and to date, no single view conveys both types of information simultaneously [12]. Hence, visualisations must either concentrate on representing code in terms of the original layout, or focus on the underlying concepts and constructs within the code. However, there is a tradeoff in terms of abstraction when presenting this information. We believe that a visualisation which uses the original layout of the code is less abstract compared to a visualisation that represents the underlying concepts and constructs using an abstract metaphor.

Ordering

An important aspect when working with source code is to know the order in which code statements (functions, methods, loops, etc ...) are executed. Further, as developers become familiar with the code, a mental model starts to arise corresponding to the ordering of these statements, in terms of lines of code. The ordering is typically created in a top down manner, when statements declared at the beginning of a document are executed before statements declared at the end of the document. There are some exceptions to this, i.e. asynchronous functions, but these are not that important at the current state of this work. We believe that this ordering might be important to visualisation designers. When visualising the various features within the code, keeping the same order as in the textual view of the code could facilitate the link between the code and the visualisation.
Characters/token information

Each statement within a code document has a specific size, in terms of characters or tokens. This might be an important aspect when creating visualisations, as the size of each statement could convey important information. Hence, if a visualisation takes the size of these constructs into account, then it could facilitate the coupling between visualisation and code.

7.2.2 Abstraction Levels

Taking the abstraction characteristics described above into account, the visualisation prototypes within the design space of integrated source code visualisation can be split into different categories based on their level of abstraction from the textual representation of the source code (using the abstraction metrics). We have established 5 main levels of abstraction, and we illustrate the levels of abstraction in Figure 7.4 by categorising the visualisations presented in Figure 7.3 into these levels. The 5 abstraction levels are explained below.

Abstraction Level 0

Visualisations that belong to this category abstract a minimum amount from the textual view of the code. This means that the viewer should be able to visually recognise that the visualisations represent a source code document. No additional information is encoded in the visualisations other than the layout of the code, colours corresponding to the syntax highlighting within the original document, the location of the cursor, and the lines of code that are currently visible on the display medium. An example of such a visualisation is shown in Figure 7.3 - A and Figure 7.3 - B. Further, a more detailed example is shown in Figure 7.5.

The visualisation shown in Figure 7.5 can be described as a code mini-map. The visualisation is present in many source code editors and integrated
development environments, including atom and sublime text. This type of visualisation presents viewers with a zoomed out representation (overview) of the currently open source code document.

The code mini-map visualisation is based on the code-map metaphor and acts as an overview component integrated into a source code editor that maps source code to a zoomed out representation, either by the use of pixels, pixel lines, or a scaled representation of text. The visualisation explicitly shows the original layout of the code, which is an important aspect as the layout often conveys a great deal of information [48]. As the visual representation can be described as a zoomed out view of a source code document, we believe that this type of visualisation can be seen as the lowest level of abstraction from the original textural view of the code. Further, as the visualisation shows the original layout of the code, we believe that this forms a tight coupling between visualisation and textual view of the code. This makes the visualisation easy to understand, therefore yielding high levels of trust on the part of the user [49].

**Abstraction Level 1**

Visualisations belonging to this level of abstraction within the design space of integrated source code visualisation retain the familiar layout of the code, in terms of a zoomed out representation. However, the visualisation is augmented, for example with a layering mechanism, to encode various microscopic and macroscopic details within the code. An example of such a technique is shown in Figure 7.6 which adds an overlay to the code mini-map visualisation, in the context of encoding the scope chain hierarchy and scope relationships within source code. The code mini-map, can therefore, be used to encode microscopic and macroscopic details that are present within the currently open source code document. But, we see this as a more abstract
way to visualise information compared to the original code mini-map belonging to Abstraction Level 0. We refer to the shift from Abstraction Level 0 to Abstraction Level 1 as a semantic turn as we move from representing simple microscopic syntactic details within the visualisation to representing higher level macroscopic, or semantic, information. Depending on the task at hand, a layering mechanism could be added to the code mini-map, where each layer adds a specific overlay that encodes only a certain property of the code.

**Abstraction Level 2**

Visualisations belonging to this level of abstraction mainly focus on the underlying constructs and concepts present within a source code document. Therefore, the layout of the code is no longer apparent in the visualisation. The underlying constructs and concepts within the code are extracted from
7.2. The Design Space

The code and used to generate a visual representation. Each extracted construct is represented as a geometric shape (e.g. each rectangle represents a scope or control structure). The top to bottom ordering and flow of the constructs is respected. This means that the shapes within the visualisation correspond with the ordering of constructs extracted from the source code. Further, to link the visualisation to the textual representation of the code, the number of tokens (other information can also be used e.g. line/column indexes) corresponding to each construct are counted and the size of each geometric shape encodes the number of tokens that are part of that construct within the code. The start and end token index of each construct is used to calculate the position of each geometric shape. This means that the positions of each shape also corresponds to the positions of the construct in the textual view of the code, in terms of lines numbers. Figure 7.7 presents an example of a visualisation from this abstraction category.
Abstraction Level 3

Visualisations belonging to this level of abstraction encode the underlying constructs and concepts present within a source code document. Geometric spaces are used to represent these underlying concepts and constructs. The size of each geometric shape no longer corresponds to the number of tokens/characters that a specific construct is composed of. Further, the positions of each geometric shape no longer correspond to the lines of code on which the constructs start/end. However, the ordering of the geometric shapes still corresponds to the ordering of the constructs within the source code document. Figure 7.8 presents an example of a visualisation from this abstraction level category.
Abstraction Level 4

The ordering of underlying constructs as it occurs in source code document is no longer used when creating the layout of the visualisations belonging to this abstraction level. The position of each node is calculated based on the layout algorithm and not on the original order within the source code document. The size of each geometric shape is the same, as token information is no longer used. Figure 7.9 presents an example of a visualisation from this abstraction level category.

7.3 Summary

In this section we establish a design space for integrated source code visualisation. We define the use of integrated source code visualisation to describe visualisations that are meant to complement existing software development
tools (source code editors and integrated development environments). The main goal of the design space is to drive exploration of various different types of visualisations designed to represent source code. These visualisations can in turn be used by software developers to facilitate the process of working with source code, more specifically working with a source code document.

A series of prototype visualisations are also presented, all of which belong to the newly established design space of integrated source code visualisation. These visualisation prototypes can be split into several categories, based on their level of abstraction from the original source code document. We denote several aspects by which the level of abstraction of a visualisation can be measured, in the context of a source code document. Further, we defined, described, and presented a series of examples corresponding to the different levels of code abstraction.

An open question related to the design space of integrated source code visualisation...
visualisation is how the level of abstraction of a visualisation effects the understanding of a programmer, when faced with the task of working with source code. We address this question, in the form of an experiment. More details can be found in Chapter 8.
Chapter 8

From Concrete to Abstract

In Chapter 7 we introduced a design space for integrated source code visualisation and presented several prototype visualisations in combination with a source code editor. In Chapter 5 we described the code mini-map and reported an experiment to investigate its effect on code understanding. Additionally, in the Chapter 6 we presented Scoped, a prototype tool that uses a tree visualisations technique to encode the scope chain hierarchy and a list view to present information related to the scope chain hierarchy within a source code document. An experiment which investigates the effect of Scoped on code understanding is also presented in Chapter 6. The Code Mini-Map and Scoped are used to represent similar information, however the level of abstraction from the original textual view of the source code differs.

In this chapter we present four prototype software visualisations (one similar to the code mini-map, and an other similar to Scoped), all of which belong to the design space of integrated source code visualisation and conduct an experiment which measures the effect of these prototypes on code understandings. The main difference between the prototypes is the level of abstraction from the original textual representations of the code. Hence, we are interested in analysing the effect of a visualisations level of abstraction on code understanding.
8.1 Prototypes

All of the prototype visualisations depicted in this section focus on encoding scope relationships within source code. However, we believe that these visualisations can also be used to represent other details within a code document (control structures, variable declarations, etc ...) The goal of these visualisations is to help programmers answer questions such as:

1. In which scope is the cursor of the source code editor currently located in?

2. To which scope does a specific variable or function belong to?

Figure 8.1 depicts a source code fragment written in JavaScript, where the cursor is located on line 14. By reading the code, we can figure out that the cursor is currently located in the scope of the for loop declared on line 8. Further, even though the variables, word and firstLetter are declared in the scope in which the cursor is currently located, they are hoisted to the upper function scope (camelCase(str)). In the context of this example, we can see that determining to which scope a specific variable belongs and in which scope the cursor is currently located involves background knowledge of the programming language as well as a correct understanding of the underlying concepts within the programming language. Therefore, understanding these hidden structures and constructs can become quite a daunting task. We believe that integrated visualisations can facilitate the process of code understanding.

8.1.1 Prototype 1

Figure 8.2 shows the first of the four prototype visualisations. This visualisation belongs to the Abstraction Level 1 category described in Chapter 7 and can be described as a code-mini map with an additional layer of information superimposed that corresponds to scope relationships within the source
8.1. Prototypes

Figure 8.1: Sample code fragment written in JavaScript

The code mini-map reduces each line of code to one pixel line, keeping the same line layout. Additionally, a grey highlighted area is also added to the code mini-map which shows the current dimension of the source code editors viewport so viewers can see which section of the code document is currently visible on screen. More information corresponding to the visualisation technique used in this prototype can be found in Chapter 5.

The syntax highlighting present on the textual representation of the code can also be transferred to the code mini-map. As was the case with the visualisation presented in Chapter 5, the yellow highlighted area on the code mini-map encodes the extent of the scope in which the cursor is currently located. The red border on the code mini-map shows to which scope a highlighted variable belongs. Each time the cursor changes location within the code editor the code mini-map is updated. If a programmer is interested in seeing to which scope a specific variable belongs, then the programmer must highlight that specific variable within the code editor. Figure 8.2 shows an
example of this behaviour. Additionally, by placing the mouse pointer either in the yellow highlighted area or within the red border, the corresponding lines of code are highlighted within the code editor. Thus, the viewer can see which lines of code belong to either the scope in which the cursor is located in or the scope to which a specific variable belongs to. An interactive demo of the prototype can be found here: https://ivanbacher.github.io/p-016-experiment-testing/#/step_04A

### 8.1.2 Prototype 2

Figure 8.3 shows the second prototype. This visualisation is more abstract than the code mini-map, hence, it can be placed into the *Abstraction Level 2* category described in Chapter 7. The code is no longer shown using pixel lines, but the underlying constructs (scopes) are encoded as geometric shapes. Containment is used to show scope nesting. The position and size of each geometric shape correspond to the token information collected from the original source code. For example, if a scope has 30 tokens associated with it,
8.1. Prototypes

then this number is used to calculate the size. Further, if the same scope is
50 tokens from the beginning of the code document, then this information is
used to calculate the positions on the y axis.

The scope in which the cursor is currently located (Figure 8.3 line 68) is
encoded using colour. The geometric shape corresponding to that scope is
highlighted in yellow in the visualisation. The geometric shape that has a
red border corresponds to the scope to which the selected variable in Figure
8.3 belongs. An interactive demo of the prototype can be found here: https://
ivanbacher.github.io/p-016-experiment-testing/#/step_04B

8.1.3 Prototype 3

Figure 8.4 shows the third prototype. The visualisation can be placed into
the Abstraction Level 3 category described in Chapter 7. The underlying con-
structs (scopes) are encoded as geometric shapes and containment is used to
show nesting. The position and size of each geometric shape no longer correspond to the original source code and are calculated using a space filling slice and dice tree-map layout algorithm. Further, each geometric space has a bit of padding so that the hierarchical structure is clearly shown.

The scope in which the cursor is currently located (Figure 8.4 line 68) is encoded using colour. The geometric shape corresponding to that scope is highlighted in yellow in the visualisation. The geometric shape that has a red border corresponds to the scope to which the selected variable in Figure 8.4 belongs. An interactive demo of the prototype can be found here: https://ivanbacher.github.io/p-016-experiment-testing/#/step_04C

8.1.4 Prototype 4

Figure 8.5 shows the fourth prototype. The visualisation can be placed into the Abstraction Level 4 category described in Chapter 7. As with the visualisations found in prototype 2 and prototype 3, the underlying constructs

```javascript
50 = function toString(val){
51     return val == null ? "" : val.toString();
52 }
53
54
55 // Repeat string n times
56 + function repeat(str, n){
57     let result = "";
58     str = toString(etc);
59     n = toint(n);
60
61 if (n < 1) {
62     let res = "";
63     return res;
64 }
65
66 while (n > 0) {
67     if (n % 2)
68         result += str;
69     n = Math.Floor(n / 2);
70     str += str;
71 }
72 return result;
73 }
74
75 // "Convert" value into an 32-bit integer.
76 + function toInt(val){
77     // doesn’t break the functionality
78     return --val;
79 }
```
(scopes) are encoded as geometric shape. In this case the shapes are circles. The position and size of each circle do not correspond to the original source code and are calculated using a space filling circulare tree map layout algorithm. Also, the positioning of the circles does not correspond to the order in which the scope constructs are placed within the original source code document.

The scope in which the cursor is currently located (Figure 8.5 line 68) is encoded using a yellow highlighted circle in the visualisation. Further, the circle that has a red border corresponds to the scope to which the selected variable in Figure 8.5 belongs. The prototype shown in Figure 8.5 is similar to the one presented in Chapter 6, however, it does not included a detail view component. Further, both prototypes also use different colour encodings. An interactive demo of the prototype can be found here: https://ivanbacher.github.io/p-016-experiment-testing/#/step_04D
8.1.5 Interactions

All of the prototypes (prototype 1, prototype 2, prototype 3, and prototype 4) support the same interactions. Moving the cursor within the code editor will update the visualisation in the context of highlighting the current structure in which the cursor is located. If the cursor is placed over the currently selected scope within the visualisation, the corresponding lines of code are also highlighted within the code editor. Adding, deleting, or modifying existing code will also update the visualisation. As soon as the text representations of the code changes, the visualisation changes to represent the newest version of the code. Highlighting a specific variable, function, or parameter within the code editor will add a new encoding to the visualisation. This encoding represents which scope the selected variable, function, or parameter is a part of.

8.2 Experiment

The purpose of the experiment presented in this section is to explore whether there was a correlation (either positive or negative) between the level of abstraction of a source code visualisation, as categorised by the design space introduced in Chapter 7, and the effectiveness of this visualisation in improving source code understanding of a programmer. For this experiment the details that the visualisations encoded represent the scope chain and information related to the scope chain within a source code document. More information about the scope chain can be found in Chapter 2 Section 2.2. Further, the experimental methodology described in Section 3.2 was followed for this experiment.

The four prototype visualisations presented to participants of the experiment explicitly encode the same information. However, they differ in terms of the level of abstraction from the code. The experiment followed a between
subject design, where participants were randomly split into four groups (Group A, Group B, Group C, and Group D). Group A was presented with the prototype visualisation shown in Figure 8.2, Group B was presented with the prototype visualisation shown in Figure 8.3, Group C was presented with the prototype visualisation shown in Figure 8.4, and Group D was presented with the prototype visualisation shown in Figure 8.5.

Similar to the experiments presented in Chapter 5 and Chapter 6, the programming language used in this experiment is JavaScript. As a prerequisite prior to participating in the experiment, participant were asked to answer several questions in an attempt to measure their general programmer experience and JavaScript programming experience. Participants were asked to enter their self estimated experience level using a 5 point Likert scale where each number within the scale was replaced with a level of Dreyfus’ model of skill acquisition [11, p.162]: 1) Novice, 2) Advanced beginner, 3) Competent, 4) Proficient, and 5) Expert.

Figure 8.6 shows the participants’ self estimated experience level with the JavaScript programming language. The images show that the four groups have a similar distribution of novice to expert programmers. The results of a Kruskal-Wallis test show that there is a no statistically significant difference between the groups when comparing JavaScript programming experience (p-value = 0.99).

All participants were presented with the same source code documents and asked to answer a series of questions designed to interrogate their understanding of the scope chain and information related to the scope chain within those documents. The questions were designed using an analysis of popular stack overflow questions related to understanding scope in source code [1]. Participants were presented with the same set of questions. The ability of participants to correctly answer questions was used as a proxy measure for their understanding of scope within source code documents. By measuring
FIGURE 8.6: Distribution of participant programming experience
the differences in the ability of participants to correctly answer questions under different conditions we explore the effectiveness of the different levels of abstraction of the visualisations included in the experiment.

8.3 Results and Discussion

For this experiment 52 subjects were recruited and randomly split into four groups. Hence, all groups (Group A, B, C, and D) consisted of 13 participants. All participants answered 10 code understanding questions related to the concept of scope and the scope chain. To measure potential differences between the groups on an aggregated level, the number of correct answers given by each participant was counted.

Figure 8.7 presents the distribution of participant correctness scores for Group A, Group B, Group C, and Group D. The plots show that the groups have different distributions and that a greater number of participants in Groups A and B have a higher correctness scores compared to Groups C and D. Participants from Group A have an average correctness score of 7, participants from Group B have an average correctness score of 7, participants from Group C have an average correctness score of 4, and participants from Group D have an average correctness score of 5. A Kruskal-Wallis test shows that there is a statistically significant difference between the groups (p-value = 0.02). Further, a Dunns test (Table 8.1) shows that the main statistical significant differences are between Groups A and C (p-value = 0.03), Groups B and C (p-value = 0.01), and Groups B and D (p-value = 0.02).

We believe that this is an interesting finding as it shows that participants presented with the prototype visualisations that correspond to the lower levels of abstraction (determined in Chapter 7) were able to answer the code understanding questions with higher accuracy compared to participants presented with visualisations that correspond to the higher levels of abstraction.
Figure 8.7: Correctness scores split by groups
Further, the results seem to indicate that there is a step change in effectiveness between Prototype 2 and Prototype 3. So this seems to imply that the step in abstraction between L2 and L3 loses important information.

Figure 8.8 shows the percent of correct answers aggregated on a group level. By examining the image we can see that participants in Groups A and B have answered at least 60% of the questions correctly, while participants in Groups C and D have answered less than 50% of the questions correctly. The results of this plot align with the results shown in Figure 8.7.

To get a more precise and in-depth understanding of participant correctness scores (Figure 8.7) and percent of correct answers aggregated on a group level (Figure 8.8), results corresponding to each individual question were also analysed. Figure 8.9 displays a grouped bar chart, where each group of bars corresponds to an individual question and the length of each bar indicates the percentage of correct answers for that question. Further, the questions are split by group.

By examining Figure 8.9 we notice that for most of the questions, participants in Groups A and B outperformed participants in Groups C and D. Further, it seems to indicate that participants in Group A and Group B greatly outperformed participants in Group C and Group D for 6 out of the 10 questions (Q2, Q3, Q5, Q6, Q7, and Q8). After examining the source code associated with these questions more closely, we noticed that, similar to previous experiments, it incorporated a specific feature of JavaScript. The code used for these questions relied on variable hoisting, i.e. variables declared

---

**Table 8.1: Dunns test results**

<table>
<thead>
<tr>
<th>Group A</th>
<th>Group B</th>
<th>Group C</th>
<th>Group D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p = 0.65</td>
<td>p = 0.01</td>
<td>p = 0.02</td>
</tr>
<tr>
<td>Group B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group C</td>
<td>p = 0.03</td>
<td></td>
<td>p = 0.84</td>
</tr>
<tr>
<td>Group D</td>
<td>p = 0.06</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
in a block scope (if, while, for, and switch statements) are hoisted to the upper function scope. This behaviour can be a great source of confusion for novice and experienced programmers alike, especially for programmers that are mainly familiar with languages such as C, which do not implement variable hoisting. This was also observed in previous experiments which can be found in Section 5.3 and 6.3.2. This is an interesting finding, especially when comparing Groups B and C, as the visualisation prototypes given to these groups are fairly similar. A possible explanation could be that by relating the geometric shapes presented in the prototype given to Group B more closely to the original source code, in terms of token information, participants were able to link the visualisations with the original code document. This reasoning would also explain the results of Group A, as the visualisation prototype given to this group can be described as being the lowest level of abstraction from the original source code document.

In summary, the results of the experiment shows that combining visualisations with source code editors can have a positive effect on code understanding, especially in situations where the textual representation of the
code no longer corresponds to the actual behaviour of the code. The results of the experiments presented in Chapter 6 also seem verify this statement. Additionally, the results presented in this chapter show that the less a visualisation is abstracted from the original textural representation of the code, in terms of line, token, ordering, and character information, the more likely it is that viewers can link the visualisation to the code and back. We believe that this is an important finding, as software visualisation researchers and practitioners can use this information during the design, developments, and implementation of future software visualisation tools.
Chapter 9

Conclusion

The following chapter includes a brief summary of the presented work, answering the main research questions this thesis presents. Further, the chapter provides additional information that corresponds to general limitations, reflections, and future work suggestions.

9.1 Summary

Visual representations tap into the capabilities of the powerful and highly parallel human visual system [54]. Hence, given the high complexity of source code, it is reasonable to consider how visualisation can facilitate source code understanding. Hendrix et al. [31], Miara et al. [52], and Wettel et al. [75] illustrate the effectiveness of using visualisation to facilitate source code understanding.

Source code editors are designed specifically for the reading, writing, and editing of source code, which makes them a fundamental tool for software developers. In this work we illustrate that combining visualisations with source code editors can facilitate the process of understanding the various complex features within a source code document. We presented several prototype tools using various visualisations techniques. Interactive demos of these prototypes can be found in Appendix D. The prototypes are concerned
Chapter 9. Conclusion

with encoding the static structure of a source code document, as well as conceptual structures within the document. The main conceptual structure that was encoded can be described as scope and the scope chain hierarchy, however, we believe that the presented visualisations can also be used to represent other microscopic or macroscopic details within a code document.

The research questions we addressed in this thesis were:

1. Do programmers make use of source code visualisations?

2. In which cases are source code visualisations that are integrated into source code editors helpful and in which cases are they unhelpful?

3. Are there visual representations that programmers find easier to link back to the text view of the code, and if there are, what are the characteristics of these visual representations?

RQ1 was answered in Chapters 4, 5, 6, and 8. In these chapters we show that software developers do make use of source code visualisations, especially if they are integrated into source code editors. An important point to note is that by integrating software visualisation into source code editors, users are still able to view the textual representation of the code and link this representation to the visual encodings present in the source code visualisations. We observed that when attempting to answer questions related to the static structure of HTML code document or the scope chain hierarchy within a JavaScript code document, software developers (novice and experts) used the visual representations to verify assumptions or to gain a high level overview.

RQ2 was answered in Chapters 4, 5, 6, and 8. In these chapters we show that software visualisations must be tailored towards visually providing an answer to a specific question a software developer might have when dealing with source code. Hence, including a generic visualisation will not be of
9.1. Summary

much use to the viewer. Further, in Chapter 5, we also observed that an in-
appropriate visualisation can actually have a detrimental effect on task per-
formance, while at the same time giving participants false confidence that
they are performing well. An additional observation was that source code
visualisations are particularly helpful in situations where the textual repre-
sentation of the code no longer corresponds to the actual behaviour of the
code. An example of such a situation is variable hoisting within a Javascript
code document.

RQ3 was answered in Chapters 7 and 8 and inspired by information de-
derived from Chapter 2, in which we identify that software developers become
very code centric when working with source code. Hence, the more a visual-
isation can link to the original source code representation, the easier it should
be to interpret and to use. With this in mind, we created a design space for
integrated source code visualisation, presented in Chapter 7, which includes
a series of prototype visualisations that range from concrete to abstract rep-
resentations of source code. The most concrete visualisations being the ones
that link to the code in terms of code layout, ordering of code constructs,
and characters/token information. In Chapter 8, which evaluates a subset of
the visualisations within the design space of integrated source code visuali-
sation, we observed that the more abstract a visualisation is, the smaller the
positive effect on code understanding. This means that the further a visual-
isation diverts from the textual representation of the code, the harder it is to
understand and link back to the code. This includes the indentation within
the code and the top to bottom ordering of code constructs. The code mini-
map presented in Chapters 2 and 5 is a good example of a visualisation that
does not abstract to much from the textual representation of the code, hence
it yields a high level of trust on behalf of the user. We believe that in order for
a visualisation to be successful in the context of facilitating code understand-
ing, it must not abstract to much from the original layout of the code and the
ordering of the structural elements within the code.

An important aspect that we noticed during the process of creating this work was that a visualisation should be explicitly tailored to answer a very specific question a software developer might ask when working with code. This is due to the fact that developers become very code-centric during the process of writing, reading, and modifying code, and will only use a software visualisation when attempting to understand a specific property or concept within the code. Hence, we believe that the ideal situation is to create and add many visualisations, all of which encode specific aspect of the code, where a software developer can use these visualisations in a pop up manner, when attempting to answer a specific question about the code. For example, when dealing with issues related to scope and the scope chain within a source code document, a developer might ask the questions of to which scope a specific variable is part of. Hence, a simple visualisation that explicitly shows the answer to this question is of more use to the developer, than a generic visualisation that attempts to encode many different features within the code. Further, we think that these visualisations do not need to be visible at all times. Developers should be able to choose when to show/hide these visualisations, as screen real-estate is a very valuable attribute when it comes to software development.

9.2 Limitations

This following section reviews the generalisability of the prototypes and experimental results presented in this work.

Recruited Participants: Feigenspan et al. [25] report that self estimation indicates programming experience reasonably well. Indeed, responses to self estimation questions correlated to a strong degree with the experience level of participants in their experiments. Following this result, all participants in
our experiment were asked to subjectively estimate their programming experience level. We believe that our sample sizes are a balanced mix of novice to expert programmers. To mitigate the threat that participants expertise may not have been fairly distributed across the control and experimental groups, we used randomisation to assign treatments to subjects.

**Experimental Setting**: The physical or virtual location in which an experiment is conducted is an important aspect that can affect the generalisability of the results of an experiment [37]. The presented experiments were mostly conducted as remote experiments, where participants used a web browser of their choice to navigate to a webpage which hosted the experiment. Although participants’ could use their actual workspace during the experiment, these types of experiments are more difficult to control compared to experiments conducted in a lab setting. First, it is hard to control for interruptions during the experiment. To mitigate this, the elapsed time was recorded after each step of the experiment was completed. The results show that participants on average completed the experiment in the same amount of time, therefore, we assume that interruptions did not overly affect the experiment. Second, we cannot be certain that if the tool worked correctly. To avoid this issue, the tool was written using web technologies (HTML, CSS, JS) that are supported in all browsers. Additionally, the JavaScript libraries that were used are well known and thoroughly tested (e.g. D3 and jQuery). We also recorded and compared mouse cursor position data of all participants. Examining this data, we did not noticed anything out of the ordinary.

Following Ko et al. [37] suggestion that if a tool is entirely new, it may be more valuable to observe a tool being used in more realistic conditions with fewer controls to better understand the benefits of the tool in practice. We opted for a more realistic experimental setting, however, this was at the price of less control.

**Source code editor read-only mode**: The experiment design decision to
set the source code editor, which each participant used, to read-only mode was made to prevent participants for adding, modifying, and removing code needed to answer the provided questions. We understand that this can effect the real world utilisation in terms of how visualisations respond to changes within a source code document (consistency). Additionally, due to the fact that the experiment was conducted as a remote experiment, allowing participants to add, modify, or remove code would add a great deal of complexity in terms of providing a mechanism to execute the modified code.

**Questions:** A difficult aspect of controlled experiments is creating questions/tasks which reflect real world usage scenarios. In order to mitigate this threat, the results of an analysis which gathered data on programming issues related to the concept of scope by examining popular stack overflow questions were used to create a set of questions that relate to real-world programming issues.

**Generalisability:** There are many other complex features related to source code that can be visualised. While many of the prototype visualisations presented in this thesis explicitly focus on encoding scope relationships within a source code document, we believe that these can also be tailored to encode other complex aspects within the code. Further, many of the experiments presented in this thesis use source code written in JavaScript. We believe that the results presented in these experiments can be generalised to other programming languages.

### 9.3 Future Work

The main goal of this work was to investigate the use of visualisation in the context of source code understanding. We present several prototype visualisations, all of which combine existing and new visualisations techniques with
9.3. Future Work

Further, we have conducted several controlled experiments in an attempt to measure the effect of these visualisations on source code understanding and these experiments generated a series of promising results. A very important aspect we would like to highlight is that we never explicitly make the claim that the visualisations designed and created in this work are the best possible choice for visualising the various complex details within a source code document. We believe there are many possible visualisations techniques that can be used to visualise source code and the ones we have designed and implemented can therefore be used by future practitioners or researchers as baselines or as inspiration.

Figure 9.1 illustrates an additional prototype visualisation that is integrated into a source code editor. This prototype is still at an early stage of development, however, it can be seen as the successor to the prototype presented in Chapter 4 which uses an icicle tree diagram to encode the structure of an HTML document. In this approach, each visible HTML element in a
Chapter 9. Conclusion

HTML document is rendered as a 3D box. To produce the layout, the width, height, margins, borders, and padding of each HTML element are taken into consideration. Furthermore, the viewer can change a setting that corresponds to the width of the screen. This is helpful for inspecting and investigating different layout on mobile devices. Colour within the visualisation is used to encode in which HTML element the cursors within the source code editor is currently located. If the cursor is located within a non-leaf HTML element (an element with child elements), the elements contained within the parent element are drawn somewhat transparently. It is also possible to rotate the visualisation and to zoom in/out. The initial version of the prototype is located at https://ivanbacher.github.io/p-006-prototype-live/.

The 3D representation fits particularly well with the CSS Box Model, which is used when talking about the design and layout of HTML elements. Figure 9.2 shows an example of the CSS Box model. In the model, all HTML elements are considered as boxes, consisting of margins, borders, padding, and content. Content refers to the content of the box, where text and images are rendered. Padding refers to the area around the content. Border refers to the space that goes around the padding and content and margin refers to the area outside the border. The model allows developers to add a border around elements, define the space between elements, and add space around the content of an element.
Chapter 5 introduces a layering concept to the code mini-map visualisation, which can be described as a mechanism for providing additional information to the viewer, while using the same underlying visualisation. As an example of this approach, the chapter illustrates a scope information layer, which is a layer of encodings that are superimposed into the code mini-map visualisation, to facilitate a programmer’s understanding of the scope chain and information related to the scope chain within a source code document. Further, the results of the conducted experiment show that superimposing an additional layer of encodings onto a code mini-map has a positive effect on code understanding. A direction for future work entails the design, implementation, and evaluation of additional layers that can be added to the code mini-map visualisation. These layers can encode code execution hotspots, code ownership, search query results, function call hierarchies, or control structure hierarchies.

In Chapter 6 we present Scoped, an interactive visualisation tool. A demo of the tool can be found at: \texttt{http://tiny.cc/jsscope}. The results of the conducted experiments indicate that adding a composite visualisation to a source code editor can have a positive effect on code understanding and that both the overview and detail view components of the composite visualisation are important. For future work, it would be interesting to conduct similar experiments, as the ones presented in Chapter 6, using debugging tasks. Moreover, the visual encodings presented in Scoped could also be used to encode other aspects within a source code document relating to the structure, behaviour, and evolution of the code.

The design and development of different approaches which can be used to encode the scope chain hierarchy within a source code document could also be a promising direction for future work. As source code contains many
Chapter 9. Conclusion

types of relations and hierarchies. It makes sense to consider using tree visualisation techniques to encode some of these to facilitate source code understanding and navigation. Interesting visualisation techniques to consider include the use of other tree visualisation techniques, such as tree-maps [33] and icicle trees [30]. Additionally, techniques which employ the code-map metaphor [3] are worth investigating, as these could provide a more natural mapping.

As an exploration of using tree visualisation techniques to encode these hierarchies and relationships within a code document, we have developed a further prototype. Figure 9.3 illustrates this prototype, which is composed of an icicle tree diagram, a circular tree-map, and a source code editor. Nodes within the icicle tree encode the control structure hierarchy of the source code document located in the source code editor, and are represented as graphical primitives. Parent-child relations are encoded by horizontal adjacency, meaning that child nodes are placed to the right of their parent. Nodes within the circular tree-map encode the scope hierarchy of the source code document located in the source code editor, and are represented as nested circles. Parent-child relations are encoded by containment, thus, child nodes are drawn within circles representing parent nodes. Depending on which control structure and scope the text cursor is located in within the source code editor, the corresponding nodes within the icicle tree diagram and circular tree-map are highlighted.

We consider this to be a possibility for future work. As many other tree visualisation techniques exist, a promising direction for future work also includes the investigation of the appropriateness of these techniques, or a combination of the techniques, for the visualisation of the many types of relations and hierarchies within source code. An initial attempt to bring in other tree visualisation techniques can be seen in Figure 9.4 (prototype available at http://tiny.cc/pgq6by), which also shows the use of colour for encoding
control structure or scope types within the visualisations.

In Chapter 7 we establish a design space for integrated source code visualisation. The main focus of this design space is to describe visualisations that are meant to complement existing software development tools. We present a series of prototype visualisations, all of which belong to this design space and can be categorised based on their level of abstraction from the original source code document. A possible direction for future work could be the design and development of further visualisation prototypes for this design space. Further, additional categories corresponding to the design space could also be developed in which the prototype visualisations can be placed based on their level of abstraction.

Chapter 8 includes an experiment in which a subset of the prototypes described in Chapter 7, which belong to the design space of integrated source
FIGURE 9.4: Prototype visualisation - Possibilities
code visualisation, are presented. In the experiment we attempted to measure the effect of the visualisations level of abstraction on code understanding. Future work possibilities include performing additional experiments using the other prototype visualisations within the design space of integrated source code visualisation and comparing the outcomes of the experiments with the results presented in this work.
Appendix A

Main Technologies Used

The experiment framework for this research project, as well as the presented prototypes were implemented as web applications and created using web technologies such as HTML, CSS, and JavaScript Table A.1 shows a summary of the main libraries used in the implementation of the web applications.

Aurelia is a JavaScript client framework for web, mobile, and desktops. The framework leverages simple conventions which makes it easy to use. It is written with the newest version of JavaScript and integrates web components. This framework was used to create each of experiment web application.

D3.js is a JavaScript library for visualising data with HTML, SVG, and CSS. The library is used on hundreds of websites which include creating interactive graphics for online new websites, information dashboards for viewing data, and producing geographical maps. Three.js is a cross-browser JavaScript library that can be used to create and display animated 2D or 3D computer graphics in a web browser. The library uses WebGL and is hosted on Github. Both of these libraries where used to create the prototype visualisations which the experiments evaluate.

CodeMirror and Ace.js are embeddable code editors written in JavaScript. Both of these web libraries match the features and performance of native editors such as Sublime text, Vim, and Atom. Further, both come with a number of language modes and add-ons that implement more advanced editing
functionality. A rich programming API and a CSS theming system are available for customising CodeMirror and Ace.js to fit many use-cases. These JavaScript based editors were used in the experiment prototypes.

Bootstrap is an open source toolkit for developing with HTML, CSS, and JS. The toolkit lets you quickly prototype your ideas or build your entire app with, responsive grid system, extensive prebuilt components, and powerful plugins built on jQuery. jQuery is a lightweight JavaScript library. The purpose of jQuery is to make it much easier to use JavaScript on a website or web application. jQuery takes a lot of common tasks that require many lines of JavaScript code to accomplish, and wraps them into methods that you can call with a single line of code. JQuery also simplifies a lot of the complicated things from JavaScript, like AJAX calls and DOM manipulation. The jQuery library contains features such as: HTML/DOM manipulation, CSS manipulation, HTML event methods, Effects and animations, and AJAX.

Node.js is an open-source, cross-platform JavaScript run-time environment that executes JavaScript code server-side. It allows web developers to build fast and scalable network applications with JavaScript. Node.js can
be used to write both the frontend and backend of a web application in JavaScript. Koa.js is a widely used Node.js web application framework. It is a lightweight framework and provides a minimal interface for developing web applications and APIs. The features and tools provided by the framework further help programmers to accelerate development of web applications and APIs. MongoDB is an open-source, cross-platform, document-oriented database. It provides the persistence for your application data. MongoDB bridges the gap between key-value stores and relational databases. Instead of storing data in rows and columns as one would with a relational database, MongoDB stores JSON documents in collections with dynamic schemas. MongoDB can be used via the JavaScript/Node.js driver.

Okeanos global is a IaaS Service for the academic and research community that allows you to easily create virtual machines and virtual Networks. Students, professors, and researchers can use this service for free and get the full power of virtual infrastructures (computing, network, storage). In the context of the presented research project, the IaaS Service was used to host the experiment database which collected the results of each subject that participated in the experiments.
Appendix B

Experiment Consent Form

Contact Details:
Ivan Bacher (ivan.bacher@dit.ie), School of Computing, Dublin Institute of Technology

I consent to participate in this study. I have been informed that the confidentiality of the data I provide will be safeguarded. The electronic data that will be formed of the responses given will be permanently archived by the student/researcher and any sensitive data will be anonymised to prevent my identification. I am free to ask any questions at any time before and after the study. I have not been persuaded in any way to participate in this study and I understand that I may terminate my participation at any point should I so wish.

- My participation is entirely voluntary.

- I may refuse to answer any question.

- I may withdraw at any time without prejudice.

- I agree to Ivan Bacher and Dublin Institute of Technology to the storing of any data that results from this project.

- I agree to the processing of any data for purposes connected with this research.
• I have understood the description of the research that is being provided to me.
Appendix C

Experiment Questions/Tasks

C.1 Chapter 4 Experiment

1) How many child nodes does node ZMUQW contain?
2) How many descendant nodes does node GLUGL contain?
3) How many leaf nodes does node LUFMU contain?
4) What node is the closest common ancestor of nodes SPKNM and MLNAQ?
5) What node is the closest common ancestor of nodes HHOXU and PW-MUU?
6) Does node XDWUF contain more than 10 descendant nodes?
7) Is node JHXKP an ancestor node of node SQYHS?
8) How many child nodes does node DDCBN contain?
9) How many descendant nodes does node JFYUI contain?
10) How many leaf nodes does node EVZYM contain?
11) What node is the closest common ancestor of nodes NCDVG and DZFFH?
12) What node is the closest common ancestor of nodes ZGFOl and FVLVC?
13) Does node PWMUU contain more than 10 descendant nodes?
14) Is node FEMHD the parent node of node LUFMU?
15) How many child nodes does node CTZAS contain?
16) How many descendant nodes does node BYAYQ contain?
17) How many leaf nodes does node VFVHS contain?
18) What node is the closest common ancestor of nodes RECKE and JTKWS?
19) What node is the closest common ancestor of nodes HBKJR and NXIRL?
20) Does node TVYUQ contain more than 10 descendant nodes?
21) Is node YZWLO a child node of node CRKIL?

C.2 Chapter 5 Experiment

1) The variable value (line 39) belongs to ____________?
2) The variable len (line 13) belongs to ____________?
3) The variable temp (line 42) belongs to ____________?
4) The variable res (line 60) belongs to ____________?
5) The variable val (line 39) belongs to ____________?
6) The variable length (line 13) belongs to ____________?
7) The variable temp.value (line 42) belongs to ____________?
8) The variable result (line 60) belongs to ____________?
9) The variable temp (line 106) belongs to ____________?
10) The variable ctor (line 161) belongs to ____________?
11) The variable r (line 74) belongs to ____________?
12) The variable res (line 44) belongs to ____________?
13) The variable t (line 22) belongs to ____________?
14) The variable result (line 106) belongs to ____________?
15) The variable c (line 161) belongs to ____________?
16) The variable res (line 76) belongs to ____________?
17) The variable result (line 44) belongs to ____________?
18) The variable temp (line 22) belongs to ____________?

C.3 Chapter 6 Experiment

1) Move the cursor to line 65. Is the identifier countInner local to the scope of partitionHoare?
2) Move the cursor to line 25. Is the identifier temp2 local to the scope of quickSort?

3) Move the cursor to line 90. Is the identifier countswap local to the scope of partitionLomuto?

4) Move the cursor to line 210. Is the identifier lastEl local to the global scope?

5) Move the cursor to line 179. Is the identifier AM local to the scope of the array_manipulator?

6) Move the cursor to line 109. Is elem local to the scope of remove?

C.4  Chapter 8 Experiment

1) The variable value (line 39) belongs to ______________?.

2) The variable temp (line 42) belongs to ______________?.

3) The variable res (line 60) belongs to ______________?.

4) The variable temp (line 106) belongs to ______________?

5) The variable ctor (line 161) belongs to ______________?

6) The variable res (line 44) belongs to ______________?.

7) The variable t (line 22) belongs to ______________?.

8) The variable result (line 44) belongs to ______________?.

9) The variable temp (line 22) belongs to ______________?.
Appendix D

Prototype URLs
### Table D.1: Prototype URLs

<table>
<thead>
<tr>
<th>Description</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML visualisation 1 (Chapter 4)</td>
<td><a href="https://mute-pocket.aerobaticapp.com/">https://mute-pocket.aerobaticapp.com/</a></td>
</tr>
<tr>
<td>Control structure and Scope (Chapter 9)</td>
<td><a href="https://possible-thread.aerobaticapp.com/">https://possible-thread.aerobaticapp.com/</a></td>
</tr>
<tr>
<td>Code mini-map (Chapter 5)</td>
<td><a href="https://ivanbacher.github.io/p-008-prototype-live/">https://ivanbacher.github.io/p-008-prototype-live/</a></td>
</tr>
<tr>
<td>Scoped (Chapter 6)</td>
<td><a href="https://ivanbacher.github.io/p-004-prototype-live/index.html">https://ivanbacher.github.io/p-004-prototype-live/index.html</a></td>
</tr>
<tr>
<td>Experiment demo (Chapter 7)</td>
<td><a href="https://ivanbacher.github.io/p-016-experiment-testing">https://ivanbacher.github.io/p-016-experiment-testing</a></td>
</tr>
<tr>
<td>Design space experiment prototype 1 (Chapter 8)</td>
<td><a href="https://ivanbacher.github.io/p-016-experiment-testing/#/step_04A">https://ivanbacher.github.io/p-016-experiment-testing/#/step_04A</a></td>
</tr>
<tr>
<td>Design space experiment prototype 2 (Chapter 8)</td>
<td><a href="https://ivanbacher.github.io/p-016-experiment-testing/#/step_04B">https://ivanbacher.github.io/p-016-experiment-testing/#/step_04B</a></td>
</tr>
<tr>
<td>Design space experiment prototype 3 (Chapter 8)</td>
<td><a href="https://ivanbacher.github.io/p-016-experiment-testing/#/step_04C">https://ivanbacher.github.io/p-016-experiment-testing/#/step_04C</a></td>
</tr>
<tr>
<td>Design space experiment prototype 4 (Chapter 8)</td>
<td><a href="https://ivanbacher.github.io/p-016-experiment-testing/#/step_04D">https://ivanbacher.github.io/p-016-experiment-testing/#/step_04D</a></td>
</tr>
</tbody>
</table>
Bibliography


Andrew J. Ko and Jacob O. Wobbrock. “Cleanroom: Edit-time error detection with the uniqueness heuristic”. In: *Proceedings - 2010 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2010 Figure 1* (2010), pp. 7–14. ISSN: 1943-6092. DOI: 10.1109/VLHCC.2010.11.


