



# LUND UNIVERSITY

## Towards the Humanisation of Programming Tool Interactions

McCabe, Alan

2023

[Link to publication](#)

*Citation for published version (APA):*

McCabe, A. (2023). *Towards the Humanisation of Programming Tool Interactions*. Lund University.

*Total number of authors:*

1

### General rights

Unless other specific re-use rights are stated the following general rights apply:

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: <https://creativecommons.org/licenses/>

### Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117  
221 00 Lund  
+46 46-222 00 00

# **Towards the Humanisation of Programming Tool Interactions**

**Alan T. McCabe**



---

Licentiate Thesis, 2023

Department of Computer Science  
Lund University

ISBN 978-91-8039-775-9 (print)  
ISBN 978-91-8039-776-6 (electronic)  
ISSN: 1652-4691  
Licentiate Thesis 2, 2023  
Department of Computer Science  
Lund University  
Box 118  
SE-221 00 Lund  
Sweden

Email: `alan.mccabe@cs.lth.se`

Typeset using  $\text{\LaTeX}$   
Printed in Sweden by Tryckeriet i E-huset, Lund, 2023

© 2023 *Alan McCabe*

## Abstract

Program analysis tools, from simple static semantic analysis by a compiler, to complex dynamic analyses of data flow and security, have become commonplace in modern day programming. Many of the simpler analyses, such as the aforementioned compiler checking or linters designed to enforce code style, may even go unnoticed or unconsidered by most users, ubiquitous as they are. Despite this, and despite the obvious utility that such programming tools can provide, many warnings provided by them go unheeded by programmers most of the time.

There are several reasons for this phenomenon: the propensity to produce false positives undermines confidence in the validity of warnings, the tools do not integrate well into the normal workflow of the developer, sometimes the warning message is simply too esoteric for most users to understand, and so on. A common theme can be drawn from these reasons for ignoring the often-times very useful information given by a programming tool: the tool itself is *difficult to use*.

In this thesis, we consider ways in which we can bridge this gap between users and tools. To do this, we draw from observations about the way in which we interact with each other in the most basic human-to-human context. Applying these lessons to a human-tool interaction allow us to examine ways in which tools may be deficient, and investigate methods for making the interaction more natural and human-like.

We explore this issue by framing the interaction as a "conversation" between a human and their development environment. We then present a new programming tool, Progger, built using design principles driven by the "conversational lens" which we use to look at these interactions. After this, we present a user study using a novel low-cost methodology, aimed at evaluating the efficacy of the Progger tool. From the results of this user study, we present a new, more streamlined version of Progger, and finally investigate the way in which it can be used to direct the users attention when conducting a code comprehension exercise.





## List of Publications

This thesis is comprised of an introductory section followed by four papers. The introduction serves to give an overview of the research topic, and summarises the contributions of each paper.

### Included Publications

- **Paper 1:** *"Breaking down and making up - a lens for conversing with compilers."* Luke Church, Emma Söderberg, and Alan T. McCabe, 2021. Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group (PPIG 2021).
- **Paper 2:** *"Progger: Programming by Errors (Work In Progress)"* Alan T. McCabe, Emma Söderberg, and Luke Church, 2021. Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group (PPIG 2021).
- **Paper 3:** *"Visual Cues in Compiler Conversations"* Alan T. McCabe, Emma Söderberg, Luke Church, and Peng Kuang, 2022. Proceedings of the 33rd Annual Workshop of the Psychology of Programming Interest Group (PPIG'22).
- **Paper 4:** *"Influencing Attention In Code Reading: An Eye-Tracking Study"* Alan T. McCabe, Diederick C. Niehorster, and Emma Söderberg, 2023. Accepted for presentation at the 34th Annual Workshop of the Psychology of Programming Interest Group (PPIG'23).

### Related Publications

- *"User-Centric Study and Enhancement of Python Static Code Analysers"* Steven Chen, Emma Söderberg, and Alan T. McCabe, 2023. Accepted for presentation at the 34th Annual Workshop of the Psychology of Programming Interest Group (PPIG'23).

## Contribution Statement

All papers included in this thesis were collaborative efforts co-authored with other researchers. The authors contributions are as follows:

### Paper 1

Emma Söderberg contributed the discovery of key pieces of literature for the shaping of this paper. From these, all authors (Luke Church, Emma Söderberg, Alan T. McCabe) participated in conceptualisation of the conversational lens and the further investigation of relevant literature. Luke Church wrote the first draft of the manuscript, with the exception of Section 5, the first draft of which was contributed by Emma Söderberg and Alan T. McCabe. All authors contributed to reviewing and revising the paper.

### Paper 2

Paper 2 is closely related to Paper 1, therefore all authors (Alan T. McCabe, Emma Söderberg, Luke Church) participated in conceptualisation and literature investigation. All authors contributed to the implementation of the Progger front end client, and Alan T. McCabe implemented the server. Emma Söderberg added additional tracing functionality to JastAdd, as well as technical guidance in making use of this feature. All authors contributed to writing the manuscript, with the bulk of the first draft written by Alan T. McCabe and Emma Söderberg. All authors contributed to reviewing and revising the paper.

### Paper 3

The café study was conceived of and designed by the first three authors (Alan T. McCabe, Emma Söderberg, Luke Church). Preparation for the study was undertaken by Alan T. McCabe, including the synthesising of code problems, and the data collection was done by Alan T. McCabe. Transcription was then completed by Alan T. McCabe, and a thematic analysis of the transcripts made by Alan T. McCabe and Emma Söderberg. All authors (Alan T. McCabe, Emma Söderberg, Luke Church, Peng Kuang) took part in the design ideation session, which was driven by Luke Church. Subsequent storyboards were made by Alan T. McCabe, and the chosen storyboard was selected by all authors. Alan T. McCabe implemented the subsequent changes to the Progger tool. An initial draft of the manuscript was prepared primarily by Alan T. McCabe with assistance by Emma Söderberg, with the exception of the Background section which was written by Peng Kuang. All authors contributed to reviewing and revising the paper.

**Paper 4**

All authors (Alan T. McCabe, Diederick C. Niehorster, Emma Söderberg) contributed to designing the study. Alan T. McCabe and Diederick C. Niehorster developed the study protocol, and Alan T. McCabe created the stimuli and implemented the study protocol in Tobii Pro Lab. The study was conducted by Alan T. McCabe. Data analysis was carried out by Alan T. McCabe and Diederick C. Niehorster. A first draft of the manuscript was prepared primarily by Alan T. McCabe, with Diederick C. Niehorster contributing to the Background and Data Analysis sections. All authors contributed to reviewing and revising the paper.



## Acknowledgements

Although many different people have supported me in different ways throughout my academic journey, I would like to thank a few of them specifically.

First, I would like to thank the Swedish Foundation for Strategic Research (SSF) for funding my research.

I would like to offer my sincere gratitude to my principle supervisor, Emma Söderberg for all of the support and guidance, personal and professional, that she has provided me over the past few years. Her limitless energy and enthusiasm is an inspiration, and she has been instrumental in shaping my research, and my development as a researcher.

I would also like to thank my co-supervisor, Görel Hedin, for her wisdom and guidance throughout my studies. Additionally, I would like to thank Luke Church for supporting me throughout my research. Luke's passion for the field of HCI has kindled a great interest in me for something that I had never even heard of before starting my PhD studies, and he has had a major influence over my direction of research.

To all of my colleagues at the computer science department, particularly the SDE research group and the NEX group, I would like to give thanks for all the help along the way and for making the department a great place to work. I would like to thank my friends and family for keeping me sane, and for pretending to be interested when I tell them about my research.

Lastly, I would like to give my deepest gratitude to my beloved partner, Aga. She is the entire reason I began pursuing the path of academia in the first place, and without her endless support and willingness to pick me up when I stumble I would never have made it this far.

*Thank you!*



# CONTENTS

---

<b>Introduction</b>	<b>1</b>
1 Introduction . . . . .	2
2 Conversations and Interactions . . . . .	5
3 Conversations and Programming Tools . . . . .	8
4 Programming Interactions as a Conversation . . . . .	10
5 Conclusions and Future Work . . . . .	29
 <b>Included Papers</b>	 <b>39</b>
<b>I Breaking Down and Making Up - A Lens For Conversing With Compilers</b>	<b>41</b>
1 Introduction . . . . .	41
2 Aspects of Conversations . . . . .	42
3 Frame: Interaction as a Conversation . . . . .	46
4 Conversations with Compilers . . . . .	48
5 Prototype: Mitigating Breakdowns in Compiler Interaction . . . . .	53
6 Discussion and Implications for Future Work . . . . .	55
7 Acknowledgements . . . . .	56
References . . . . .	56
 <b>II Progger: Programming by Errors (Work In Progress)</b>	 <b>61</b>
1 Introduction . . . . .	61
2 Background . . . . .	63
3 The Progger Prototype . . . . .	68
4 Discussion and Implications for Future Work . . . . .	72
5 Acknowledgements . . . . .	73
References . . . . .	73



<b>III Visual Cues in Compiler Conversations</b>	<b>79</b>
1 Introduction . . . . .	79
2 Background and Related Work . . . . .	81
3 User Study . . . . .	84
4 Design Iteration . . . . .	89
5 Discussion . . . . .	93
References . . . . .	95
<b>IV Influencing Attention in Code Reading: An Eye-Tracking Study</b>	<b>99</b>
1 Introduction . . . . .	99
2 Background . . . . .	101
3 Method . . . . .	101
4 Results . . . . .	107
5 Discussion . . . . .	109
6 Acknowledgements . . . . .	110
References . . . . .	111

---

# INTRODUCTION

---

## 1 Introduction

In contemporary software development, tools have become ubiquitous at every stage of the development process. In the context of this work, we consider a "developer tool" to be any computer program which may assist in any activity related to programming. These tools exist across multiple levels of abstraction within the software development process, with the most concrete being at the level of actual programming and the most abstract being collaboration and organisational tools.

In this thesis, we will discuss tools used by programmers: code editors and analysis tools. The most integral of these are compilers, which conduct semantic checks on code to ensure it is properly formed, and then translate it into bytecode to allow it to run. Developers may also make use of static analysis tools such as SpotBugs<sup>1</sup> and SonarQube<sup>2</sup> which can be used to analyse semantically correct code for known bug patterns that can cause issues beyond compile time. "Static" analysis is analysis that can be made without running the program, however dynamic analysis tools exist which analyse code at run-time, although these tools are outwith the scope of this thesis. Various different programming tools, such as code editors and analysis tools, are often combined together into a single integrated development environment (IDE) for ease of use. An example of a modern IDE can be seen in Figure 1. These programming tools provide a myriad of benefits, and are often explicitly integrated into development practices at large companies such as Google [Sad+15].

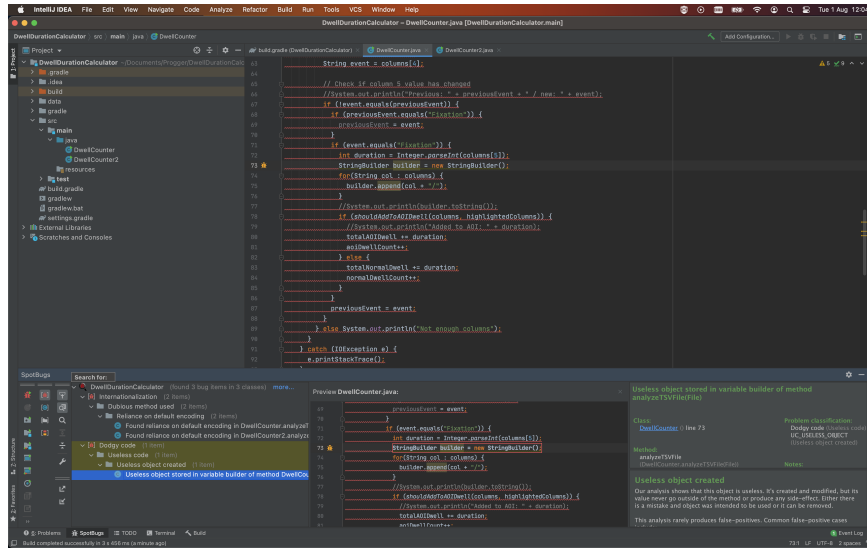
Despite the obvious utility of analysis tools in being able to point out bugs and vulnerabilities, the results of their analyses are commonly not put to use [Joh+13]. The reasons for this under-use have been explored in past-research, with a number of interesting findings. One such reason is that false positives are often commonplace, where lines are marked with a warning despite the fact that they do not contain the issue for which they were flagged. Result understandability is also a cause of frustration, with users often finding warning messages too confusing or vague to act upon the information [Bec+19]. A final such concern, although this is by no means an exhaustive list, is that the tools tend to integrate poorly into the developer workflow, forcing them to switch contexts to run the analysis and taking them out of the flow of programming.

A common thread can be drawn through these reasons for not engaging with analysis tools: usability. False positives clutter up the user interface and make results less trustworthy; esoteric warning messages require the user to dedicate time to investigating the warning message before they can even begin investigating how to fix the problem itself; poorly integrated tools force users to switch to a different application to make use of them, rather than being available directly in their development environment. Though we chose not to directly address these specific issues they ultimately became the basis for choosing the direction of this thesis, the

---

<sup>1</sup><https://spotbugs.github.io/>

<sup>2</sup><https://www.sonarsource.com/products/sonarqube/>



**Figure 1:** The IntelliJ IDE. The IDE displays a package explorer (middle left), a code editor (middle right), and the results of a static analysis conducted by SpotBugs (bottom).

primary goal of which is to investigate the analysis and design techniques which may be used for developing more usable programming tools.

In searching for a way to make interactions more natural, we delved into the mechanics of what we believe is the most innately *human* way in which we communicate information: conversation. Conversations, the act of speech between multiple human participants, predate all written methods of communication. Our most ancient extant written works, such as Homer's Illiad, are thought to have come not from an initial written manuscript, but rather from the transcription of an oral composition [Wes11]. Indeed, oral tradition is responsible for vast swathes of folklore, mythology, and cultural canon. For millennia before the advent of writing, ideas and history were shared through speech and conversation.

In this thesis, we present first our investigation into the mechanical aspects of conversation. From this research, we then developed a so-called "conversational lens" as a way of viewing a given interaction as though it were a conversation between equal parties. This allowed us to deconstruct the interaction between a human programmer and their programming tools, with particular focus on the compiler, and analyse the shortcomings of the computer in the context of a conversational partner. From this analysis, we developed our own programming tool, named Progger, drawing on the principles of conversation in its design. We then

conducted a user evaluation of Progger, the results of which further informed the development of an updated version of the tool. Finally, we investigated the use of eye movement and gaze attention as a means of non-verbal communication in conversations, and how this may be applied to Progger.

Each of the presented papers draws from the core idea of conversation as a basic unit of interaction, and uses this archetype to explore techniques for making the interaction with programming tools more "human". We finish this introductory section by presenting our conclusions and possibilities for future work based on this approach.

## 2 Conversations and Interactions

The idea of applying a conversational lens to a non-human interaction is not a novel one. The initial genesis of conversational theory, defined by Pask [Pas76], was based on his understanding of cybernetics, and was an attempt to model the learning process through the framework of a conversation. This idea has been expounded upon by various authors, particularly by Dubberly and Pangaro [DP09], who define the structure of a conversation and propose methods for applying conversational theory to the design of interactions. As outlined by Dubberly and Pangaro [DP09], conversation in its most basic form consists of a number of distinct stages:

- **Opening of a channel:** where participants in the conversation establish the grounds for communication. For example, one person (Participant A) asking another (Participant B) the question "How are you doing?" clearly defines the communication channel as verbal English.
- **Commitment to engage:** the second participant acknowledges the opening of a communication channel and indicates their receptiveness to a conversation. This takes the form of an indication that attention is being paid to the conversational partner, and usually the beginning of a turn-taking exercise, either explicitly or implicitly. E.g., Participant B responds to the question of Participant A with the statement: "I'm good, thanks. How are you?"
- **Construction of meaning:** the participants must now work collaboratively to establish the overall meaning of the conversation. This meaning making can occur from the ground up with explicit transfer of information, however it may also draw upon context given by a number of factors such as location, previous interactions, social norms, etc. If the previous conversation were to continue with the question "Did you see the football match last night?", the meaning constructed from this sentence would differ greatly based on geographical location. A pair of colleagues in Liverpool having a conversation the morning after the local derby would have a vastly different mental model to people in the USA on the morning after the Super Bowl. In this instance, the meaning differs not just by the specific match but even in the sport itself.
- **Evolution:** in a meaningful conversation, the internal states of the participants evolve throughout the process, affecting a change upon them between the beginning and ending of the conversation. This may be trivial, for instance in the expansion of a relatively inconsequential knowledge base: "No, I didn't see the match. What happened?" "Liverpool won 2-0." From this exchange, the participants evolve in the following ways:

- Participant A now knows Participant B did not watch the football match. They can therefore deduce that it is pointless asking for opinions on the quality of play or discussing specific incidents.
- Participant B now knows the result of the football match: Liverpool won 2-0.

The internal evolution of participants may be much more significant, however. Significant discoveries can be made during this process, leading to important decisions, the changing of relationships, or even the altering of entire systems of belief.

- **Convergence on agreement:** this stage describes the process of participants coming to a mutual understanding. To do this, they continue the turn-taking exercise, perhaps asking questions with the aim of clarifying each other's positions, and the exchanging of more ideas and information. As in other phases of the conversation, agreement may range from the relatively mundane - perhaps the participants agree that with their latest win, Liverpool are now well placed to win the league - to the deeply significant, such as an alignment of spiritual beliefs.
- **Actions or transactions:** a conversation often closes with the decision to take some action based upon the outcome of the discussion, for example deciding to watch the next football match together in the pub.

With this basic understanding of conversational structure established, we continued our investigation into the various aspects of a conversation. There are a number of such aspects, detailed descriptions of which can be found in **Paper 1**, however in the interest of brevity only the most salient will be explained here.

## 2.1 Breakdowns

As can be seen from the model of conversation stages, a key facet is the way in which participants come to a shared understanding. Often components of this understanding are implicit, and we sacrifice precision for the sake of speed and fluidity. An overly specific and clunky sentence such as "Did you see the association football match between Liverpool FC and Everton FC that was played last night, the 13th of February?" is condensed down to "Did you see the football match last night?" In fact, even "football" and "last night" may be considered extraneous in this example, with perhaps the maximally concise question being simply "Did you see the match?".

In being so economical with words, however, it is inevitable that misunderstandings occur. Assumptions can be mistaken and mental models can diverge, leading to confusion as participants are no longer "on the same page". We previously illustrated how context can have a significant effect on the understanding of

a statement: consider if in our example Participant B held the mistaken assumption that Participant A was in fact an avid tennis fan, and the Australian Open had just concluded. Given this context, the "match" in question may be understood as entirely different events by each participant.

With a misunderstanding as severe as this, the conversation may encounter a "breakdown". At this point, no constructive continuation can occur and participants must either abandon the conversation in bemusement or acrimony, or they must attempt to *repair* the conversation [SSJ74]. One mechanism for repair is that of a "meta-conversation" [DP09], wherein participants enter into a conversation about the conversation itself, clarifying details and assumptions until the cause of the breakdown can be identified and resolved.

## 2.2 Side channels: Gaze

Understanding in conversation is also developed through means other than words. These so-called *side-channels* are varied, and include factors such as tone and cadence of speech, body language and expression, and the direction that someone is looking, amongst others. For our research we chose the last of these, which we shall call gaze direction, as worthy of further exploration. Looking at something can attribute it with special importance in the context of a conversation, and while gaze can be explicitly directed ("look at that!"), humans also have a tendency to follow another person's gaze without instruction.

This behaviour begins developing at a very young age [FBT07], and holds an important role in social relationships. When two people are looking at the same thing, it is indicative of an aligned "joint attention". It can be safely assumed that both are consciously observing the object of attention, and any changes to its state will be registered. When considering a conversation, this means that when the participants are both looking at something it can now be considered to contribute to the *context*.



### 3 Conversations and Programming Tools

With the main aspects of conversation that we are interested in now outlined, it is possible to re-frame a programming interaction as a conversation between the programmer and their programming tools. The communication channel is the code editor, analogous to speech in a human-to-human conversation, and a programming language is used rather than a natural language. In most programming interactions, the programmer has an idea of what they want the code to do. Convergence on agreement can therefore occur by way of a turn-taking exercise that takes on a written character rather than a verbal one: the programmer inputs lines of code, and the programming environment attempts to compile and, if hot loading is used, to run it.

In this conversation, agreement can be reached in terms of both *form* and *function*. When an agreement of form is achieved, the compiler contained within the programming environment<sup>3</sup> is able to validate the syntactical and semantic correctness of the code, and execute it. To reach agreement of function, the executed program must do what the programmer intends for it to do. In our research, we elected to focus on means for achieving agreement of form.

#### 3.1 Breakdowns

In this context, a *breakdown* can occur when the programmer expects the code to execute, but compilation fails: the compiler is unable to understand the form of the communication used by the programmer. This results in a compiler error message, which is indicative of the beginning of a meta-conversation. In order to repair the breakdown, the compiler signals to the programmer the section of code that is causing confusion.

If we compare what happens now with the expected actions of a human conversational partner, deficiencies begin to appear. A human partner would likely ask probing questions about the statements that they had failed to understand, whereas the compiler simply points at the statement and says, "I don't know what you mean". From this point on, the normal turn-taking operation that we expect of a conversation breaks down, and the programmer takes on sole responsibility for repairing the breakdown.

This repair mechanism is not only inconsistent with the conversational procedure, but is also often ineffective. Error messages have been studied for decades, and have consistently been found to be a source of difficulties, particularly for novices [Bec+19]. Attempts at improvement have been made, however they still adhere to the pattern in which the compiler prints its error messages and then proceeds to abdicate all further responsibility. We hypothesised that a more fruitful method for tackling this problem may lie not in re-wording the error messages,

---

<sup>3</sup>Henceforth when discussing the ability to compile a program, the term "compiler" will be used, however it should be noted that compilation usually occurs within a larger development environment.

but by instead introducing a conversational element. To do this, we sought to re-introduce the turn-taking aspect of a conversational interaction and implement a way for programmers to "ask questions" of the compiler to gain more information about its understanding of the error. This ultimately took the form of a prototype development environment, Progger. The technical background and implementation details can be found in **Paper 2**.

### 3.2 Side-channels: Gaze

In the context of a programming tool, *gaze* as a *side-channel* is easy to define and measure for the human user: the part of the screen at which they are looking is equivalent to where their gaze may fall in a human-to-human conversation. By contrast, the computer lacks anything which may be considered "eyes", making the idea of its "gaze" much more nebulous.

During the development of Progger however, it was discovered that tracing data could be used to track the sections of code that the compiler "looked at" during compilation, and how often it did so. This provided an opportunity to align the gazes of the human and compiler conversational partners, which we attempted with highlighting. In Progger 2.0, the design process for which is outlined in **Paper 3**, a "heatmap" is displayed when a compiler error occurs. This heatmap highlights the areas of code looked at by the compiler, with the highlighting increasing in intensity as the compiler passes over the same code multiple times.

Using Progger 2.0, we then conducted an eye-tracking study to determine whether the gazes of programmer and compiler were indeed aligned. The outcome of this research is presented in **Paper 4**.

## 4 Programming Interactions as a Conversation

In this section, we present our attempt to implement a programming tool, Progger, founded on the principles of conversational design. This was undertaken with the use of an iterative, cyclical methodology as outlined in Section 4.2.

### 4.1 Objectives

When considering programming tool interactions through the conversational lens, it became clear that there were deficiencies. A human partner, when faced with a conversational breakdown, will take active steps to remedy the situation. A computer partner, however, will just say "no" or "don't know" [Bla+18] and abruptly end the interaction, placing the onus of repair squarely on the shoulders of the human. We hypothesised that introducing conversational elements to the interaction could help to solve these deficiencies, however the effectiveness of this strategy was unknown. This led us to the following over-arching research question:

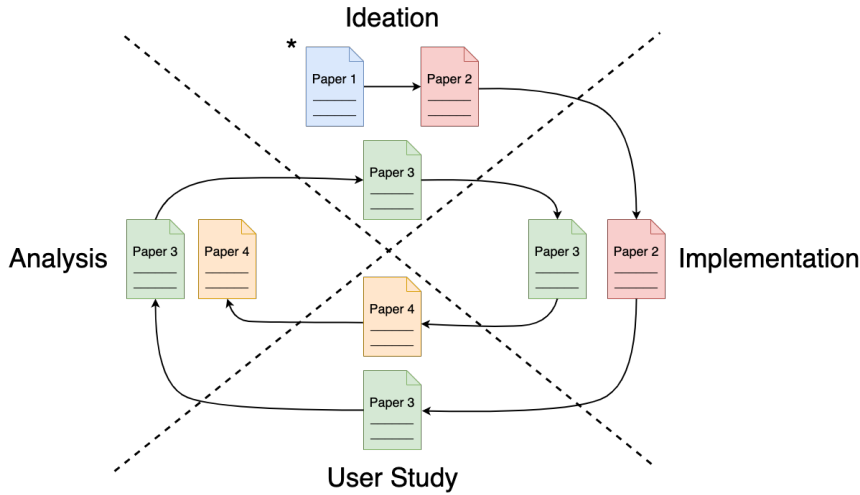
- RQ<sub>1</sub>: To what extent is the humanisation of a programming tool through conversational design beneficial to end-users?

### 4.2 Method

To address this question, we adopted a cyclical research process as shown in Figure 2. This consisted of four stages:

- **Ideation:** the collaborative process of creating a novel interaction design centred around conversational theory and previous work.
- **Implementation:** the implementation of the selected design as a programming tool.
- **User Study:** the conducting of a user study to validate the design choices and gain insight into their usage.
- **Analysis:** analysis of data gathered from the user study, with the intention that insights gained from this stage would feed into a new ideation session.

To this point, this process has been completed in its entirety twice, resulting in two distinct versions of a prototype tool and two separate user studies examining their respective efficacies. For the first cycle, following *ideation* we elected to design and build a research tool (*implementation*) for use in a *user study* and subsequent *analysis*. The initial focus of intervention was in the domain of breakdowns, which manifest as errors in the programming environment as related in Section 3.1. For this reason, the tool was named "Programming by Errors", which ultimately coalesced into the name Progger. This research tool would take the form



**Figure 2:** The cyclical process of research methodologies used throughout this thesis. The asterisk (\*) represents the starting point of the thesis work.

of a simple code editor for use with the Java programming language, which would contain additional features for the resolution of breakdowns. In order to offer more information about compiler errors, we decided to first build an extension of a Java compiler. In order to achieve detailed tracing, we extended a compiler based on the reference attribute grammar formalism, which is described in Section 4.3.

Following the first cycle of research, we undertook a second cycle with the initial *ideation* process informed by our previous work. This again led to further *implementation* work on Progger, producing Progger 2.0, and a new *user study* and *analysis*, this time with a focus on gaze. The positioning of each paper within the stages of this cycle is illustrated in Figure 2.

### 4.3 Reference Attribute Grammars

Reference Attribute Grammars (RAGs) are an extension of the Attribute Grammar (AG) formalism introduced by Knuth [Knu68]. AGs are a means of extending abstract syntax trees (ASTs), which are data structures that act as models of a program. This extension takes the form of attaching attributes to the AST nodes, where each attribute contains a formula for calculating its value. RAGs extend this with the addition of *reference attributes* [Hed00] which can hold references to other attributes located at an arbitrary distance in the AST. A number of different

attribute types can be specified. In the following list, **bold monospaced** text indicates AST nodes, and `monospaced` text indicates attributes or their equations attached to the nodes:

- *Synthesized attributes*: attributes which have a value computed from nodes *lower* in the AST. For example, an **Add** node may contain two child nodes, **Left** and **Right**, each of which contains an attribute denoting their value as the natural number 1. The value of the **Add** node may be derived from the equation: `getLeft() + getRight()`, which would resolve to 2.
- *Inherited attributes*: attributes which have a value computed from nodes *higher* in the AST. For example, children of a **Block** node may inherit the `lookup(String name)` attribute<sup>4</sup>. Child nodes that make use of a variable name, **IdUse**, may then call `lookup(name)` in order to search for the variable declaration within the scope of the parent **Block**. If no such declaration is found, the use of the variable name is considered invalid.
- *Reference attributes*: attributes containing a reference to another attribute within the AST. For example, the `lookup` attribute mentioned above may return a reference to the variable declaration node, **IdDecl**, which can be stored in a reference attribute, `decl`, within **IdUse**. From this point on, the **IdDecl** may be easily retrieved by accessing the `decl` attribute, instead of calling `lookup` each time.
- *Collection attributes*: attributes which have a value calculated from multiple contributing nodes [MEH09]. For example, an `error` collection attribute may be stored at the root of an AST. Each time an error is encountered (e.g the occurrence of an **IdUse** without a corresponding **IdDecl**), an object is added to the `error` collection, thereby compiling a list of errors throughout the AST.

For further development, we selected the Java compiler ExtendJ [Öqv18] which is built using the meta-compilation system JastAdd [HM03], an implementation of the RAG paradigm. JastAdd includes several features which are of use when attempting to glean more information from the compiler, and which were of particular importance when selecting the compiler for extension:

- *On-demand evaluation*: as the evaluation of attributes occurs on demand in JastAdd, accessing a specific attribute results in a traversal of the AST in order to analyse each node which the origin is dependent upon [SH12]. As a consequence, when the evaluation of an attribute results in an error, we can use the evaluation tree to determine all of the AST nodes that factored into this decision.

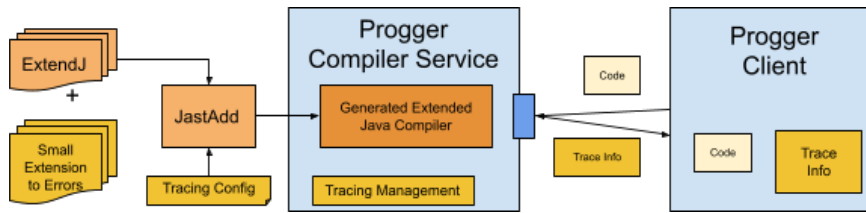
<sup>4</sup>A pattern defined by Fors et al. [FSH20]

- *Tracing*: the JastAdd tracing mechanism, first introduced by Söderberg and Hedin [SH10], allows for events to be logged and information extracted from the compiler during the aforementioned evaluation process, including details such as specific code tokens and locations attached to each node.
- *Caching*: once a given sub-tree of attributes is evaluated, the resulting value of the top-level attribute is cached, allowing for quicker compilation [SH10].

More details about the mechanics of RAGs, including worked examples, can be found in **Paper 2**.

#### 4.4 Progger Design and Implementation

For the Progger research tool, we decided to implement a web-based text editor operating on the client-server model, as shown in Figure 3.



**Figure 3:** Architecture overview of the Progger prototype. Starting from the right, the Progger Client contains a code editor that allows the user to input code. To compile, this code is transmitted via REST API to the Progger Compiler Service which contains an extended Java compiler built upon JastAdd and ExtendJ. The code is compiled with tracing information collated by the Tracing Management system. This information is then passed back to the Progger Client via REST API, where it is displayed to the user.

##### Client

The client is web page written in the Dart<sup>5</sup> programming language, with use of the CodeMirror<sup>6</sup> library to provide basic features such as syntax highlighting. Code written within the provided code editor is sent via JSON to a REST service, either by clicking a "Compile" button, or automatically after a short delay when editing the content. If a compiler error is encountered, the server responds with a JSON object representing the attribute evaluation tree<sup>7</sup> as constructed from tracing information in the compiler, with the node which threw the error as the root. This

<sup>5</sup><https://dart.dev/>

<sup>6</sup><https://codemirror.net/>

<sup>7</sup>Henceforth, the UI rendering of the evaluation tree will be referred to as simply the "trace tree".

tree is then rendered on the screen with additional UI elements in the form of a collapsible tree, with each level inviting the user to ask the compiler to "tell me more" about the error. As the user explores the tree, mousing over AST nodes which contain direct links to locations in the code results in the highlighting of the relevant section. The client with an expanded error tree is shown in Figure 4.



**Figure 4:** Screenshot of the Progger prototype showing an expanded error view. The highlighted variable (x) corresponds to the code location investigated while evaluating the attribute where the cursor is hovering.

## Server

The server takes the form of a REST service which consumes JSON objects containing Java code. Upon reception of a POST call, the content of the message is first passed to a Java 8 parser. If the code is syntactically valid, it is then passed to a custom Java compiler based on ExtendJ. At this stage, the inherent benefits of using a RAG-based compiler come into play: instead of just a call stack as would be available in a non-RAG based compiler, we are able to track the more informative attribute evaluation stack. This highlights dependencies between units of computation, grouped into *aspects* which describe rough collections of attributes and behaviour, for example "type analysis". Due to this, we can map these aspect names to stages in the compilation process and therefore assign each node in

the attribute evaluation stack a "conversational" description of the actions being undertaken by the compiler.

To gain access to the underlying information during this process, we made extensive use of the tracing system in JastAdd, including updating the JastAdd project itself to include extra trace events. The most significant of these additions was the ability to trace specific stages in the evaluation of collection attributes. This was key to our purposes, as ExtendJ uses a "problems" collection attribute, defined at the root of each compilation unit, to gather together all of the warnings and compiler errors encountered during compilation. When a contribution to this collection attribute is registered, we are therefore able to analyse the evaluation sub-tree related to this computation.

After constructing an evaluation tree for each compilation error, the trees are transformed into JSON and relayed to the client. The root node conveys the following information:

```
"message": [compiler error message],
"fileName": [file name],
"location": [line number],
"severity": [warning/error],
"rootNode": [originating attribute node]
```

Every subsequent child of the root note takes the form:

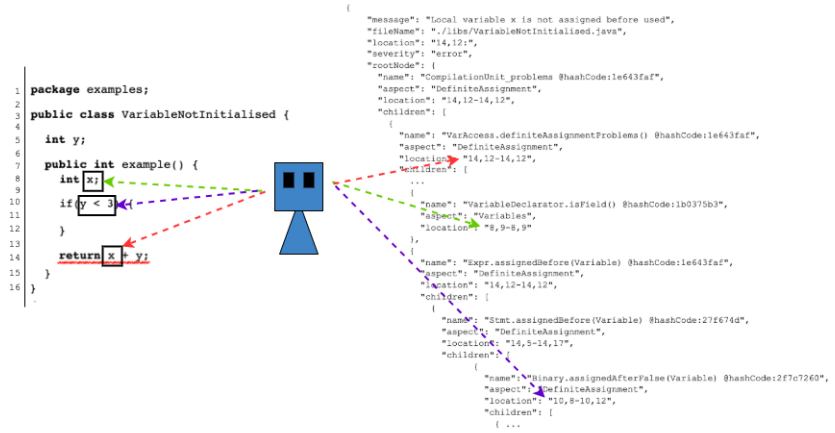
```
"name": [attribute name],
"aspect": [name of aspect the attribute is associated with],
"location": [token locations associated with the attribute],
"children": [array of child attribute nodes]
```

An example of the JSON constructed from a given error message is shown in Figure 5.

## 4.5 User Study

With an initial version of Progger implemented, we made the decision to conduct a user study. At this stage, we were not interested in the efficacy of the tool in terms of helping users to resolve errors more quickly or easily, but rather in acquiring broad sentiment data about how users felt about and used the additional information presented to them. For this reason, we deemed a full-fledged user study to be too heavy-duty and time consuming for our purposes. Instead, we opted to try a novel, light-weight and low-cost study method which was dubbed a "café study".





**Figure 5:** Example JSON excerpts of an attribute error tree. The coloured lines indicate which sections of the source code each attribute refers to.

## Café Study

The main requirement for the café study protocol was that it should facilitate rapid iteration. As a consequence, this necessitated that the study protocol be low-cost in terms of time investment for planning, set-up, and execution. As our target demographic for Progger was novice programmers (i.e. persons with little to no industrial experience, although with some experience in an academic context) we were able to make use of a large and immediate pool of potential participants: the Lund University student body. To this end, and to satisfy the requirement for at least a rudimentary understanding of programming, we decided that the location of the study should be a booth in the foyer of the E building at Lund University. Participation would be open to all students passing through the foyer, with promise of a free lunch ticket in compensation for their time. This was motivated by the knowledge that students passing through the E building would likely be enrolled in one of the science and engineering departments housed in the building, including computer science, and thus would likely have had exposure to programming in some capacity.

Study participants were invited to investigate a single-class Java program, presented in Progger, which had several different compiler errors built into the program. In total, three such programs were synthesised, based on publicly available

solutions to the Kattis programming puzzle archive<sup>89</sup>. After receiving informed consent, participants were tasked with correcting the compiler errors extant in the code. We asked participants to verbalise their thought process during this exercise, with probing questions also used to gather more information. The screen and audio were recorded for each participant for future analysis, with data collected from 13 participants in total. A picture of the booth as used in the study is presented in Figure 6.



**Figure 6:** The booth used in the Progger café study. One of the task programs is visible on the screen, with compiler errors shown on the right hand side.

## Data Analysis

After having transcribed the interviews, a thematic analysis exercise was undertaken based on the process described by Braun and Clarke [BC06]. We began by individually undertaking the task of generating a set of codes. This entailed a detailed reading of each transcript, and assigning descriptive codes for an interactions that seemed to reveal something interesting. We made a deliberate attempt to be as specific as possible when coding interactions, with a large set of codes being seen as more desirable than a smaller set of coarse-grained codes. For example, the following statements both describe the highlighting of different sections of code when exploring the Progger trace tree:

<sup>8</sup>Kattis problem archive available at: <https://open.kattis.com/>

<sup>9</sup>Solutions were taken with kind permission from the GitHub repository of Pedro Contipelli: <https://github.com/PedroContipelli/Kattis>

- Participant A: *"it's very good with the highlight system, because you know exactly where you want to look at initially"*
- Participant B: *"it's highlighting everything, it's too much"*

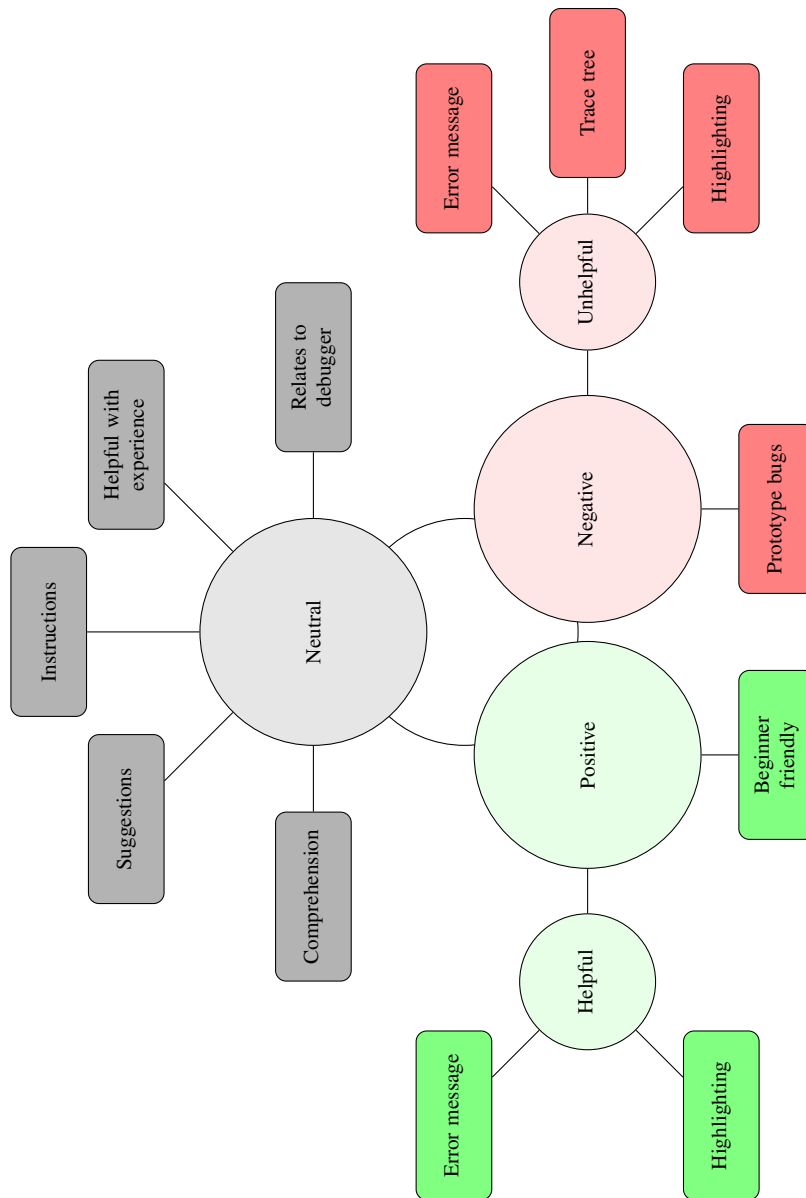
While both make reference to highlighting, they take different stances: Participant A praised the functionality as helpful in directing them to relevant areas of the code, whilst Participant B found that the amount of highlighting was confusing. For this reason, these statements were coded as "helpful highlighting" and "unhelpful highlighting", respectively. A full list of codes, including examples of coded statements and frequency of code occurrence in each transcript, can be found in **Paper 3**.

After completing the initial coding, we then met to compare and combine codes into a single unified set. Once we had settled on a unified code set, we performed a collaborative test coding of a set of 2 transcripts, in order to verify that codes were being interpreted correctly by all parties. After an agreement was confirmed, the codes were applied to the remaining transcripts individually. Throughout this iterative process, we took notice of several themes emerging in the analysis. The theme map can be seen in Figure 4.

## Results

Despite the informal nature of the café study, we found the data gathered to be of great interest. Two main findings resulted from the data:

- The code highlighting functionality offered by Progger was key to the interaction. When it worked well, for example highlighting specific lines such as variable declarations, it drew significant praise for its ability to direct the study participants to useful information. When it worked poorly, for example highlighting entire methods or even the whole class, it drew criticism for being too vague. Although the overall sentiment was positive, the mixed nature of the feedback revealed an interesting facet of the highlighting: regardless of its usefulness in each individual case, it always drew the *attention* of the user.
- The trace tree rendered in the sidebar was of no use. This was due to a general lack of compiler knowledge, making the knowledge base of the participant set (novice programmers with no industrial experience) inconsistent with the information provided by the trace tree (detailed and technical nature). Couched in terms introduced by Norman [Nor13], the gulf of evaluation was too large for the information to be useful due to difficulty of interpretation and failure to match the way in which the participants thought about the compiler.



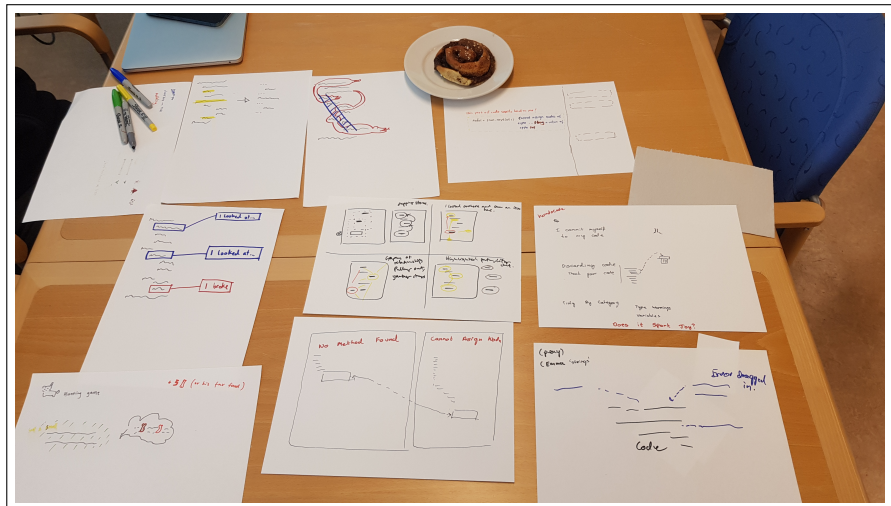
**Figure 7:** Identified **themes**, represented by circles, and associated **codes**, represented by boxes.

In these two findings, the café study effectively achieved what we had hoped for from the outset. We were able to identify components to strip out (the trace tree), and re-focus on an area that participants found of particular interest (highlighting and locality).

### Design Ideation Workshop

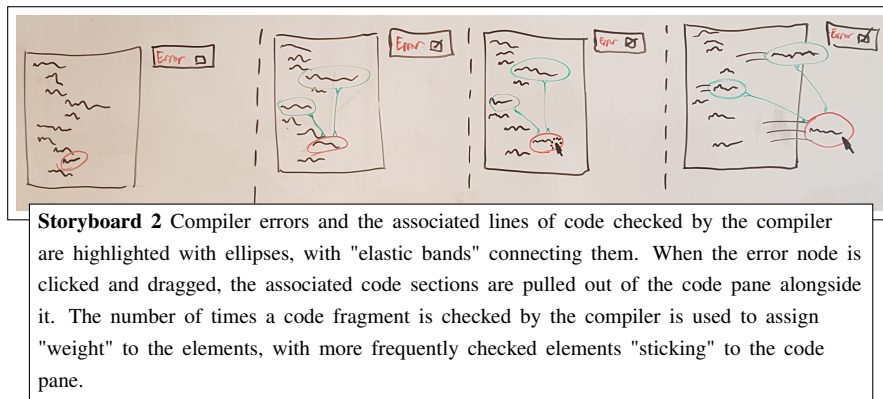
The removal of the trace tree was a simple undertaking, however the way in which a focus on locality could be operationalised was an open question. In an attempt to answer this question, we proceeded to undertake a divergent design exercise [Cro05][Dub04][Pug81] to create a large spread of new ideas for consideration. This took the form of a two-phase workshop.

During the first phase, each workshop participant sketched out a number of user interface ideas with the centre focus of the UI being code locations. It should be noted that though some of the concepts build upon the initial Progger layout of a code editor with an information sidebar, this was not a requirement and many designs diverged significantly from this formula. Following phase 1, participants took turns presenting and explaining their concepts to the group. We then proceeded to repeat the ideation process in a second phase, taking inspiration from each other's phase 1 designs, to create a final set of concepts for future development. A selection of the sketches resulting from this exercise can be seen in Figure 8, with the full set available in **Paper 3**.



**Figure 8:** A selection of design concepts as produced by the ideation workshop.

After the conclusion of the workshop, we took some time to mull over the different ideas and plan our next steps. This led to a selection of 3 candidates for further refinement, from which we produced storyboards detailing the exact process of the interaction. After discussing the storyboards, a single design was selected for implementation. All three storyboards can be found in **Paper 3**, however only the storyboard which was selected for further development is shown here in Figure 9.



**Figure 9:** Storyboard of an expanded concept from the ideation workshop. Though more storyboards were produced, this particular one was selected for further development.

The main idea behind Storyboard 2 was that when an error occurs, all relevant code sections are highlighted simultaneously, with "elastic band" elements rendered on the interface connecting the lines to the line where the error occurs. The user can then click and drag on the error line, and the connected lines are also dragged out from the main body of the code. The aspect that made this particular storyboard interesting to us was the idea that different code segments could be *weighted* differently, based on their relationship to the compiler. Specifically, we decided that elements of code would "stick" more closely to the main code pane based on the number of times that the compiler had passed over that element during the compilation process. If a section of code had been visited many times, it would move only a little, while sections of code that had been visited very few times would be dragged out alongside the error, separating them more significantly from the main body of code.

To accomplish this, it would be necessary to implement a means of assembling a compiler "heat map", where we count the number of times that each line is "looked at" by the compiler. Our interest in this particular facet of the design was born out of discussions about how exactly this information would correlate

with the underlying reason for each error. To our eyes, there were three possible results, each as likely as the other:

- Lines with a *low* heat map score (i.e few instances where the compiler looked at them) would be significant in the evaluation of the error.
- Lines with a *high* heat map score would be significant in the evaluation of the error.
- There would be *no correlation* between heat map score and relevancy to error evaluation.

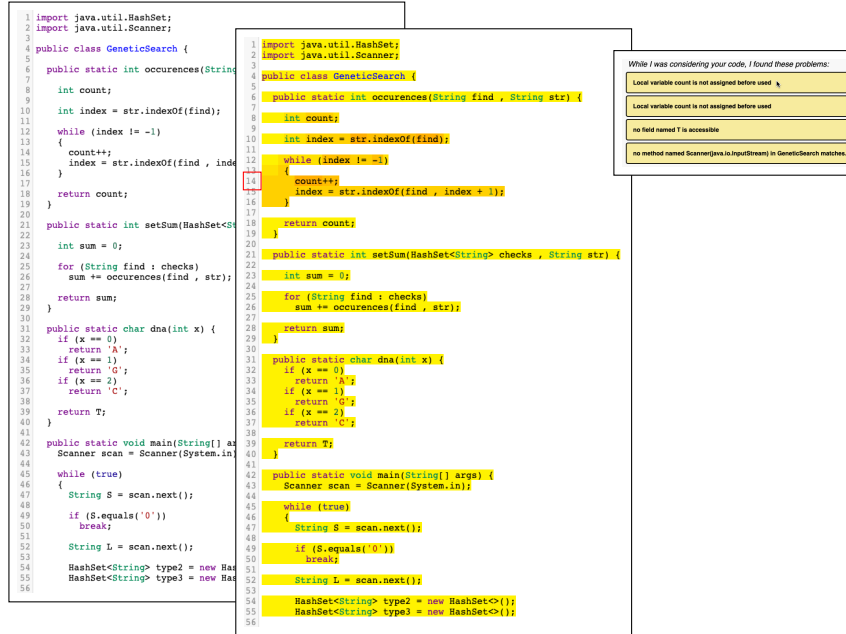
We were unable to come to a consensus as to which of these outcomes were most likely, and this ultimately proved a deciding factor in the selection of Storyboard 2. With this decision, we started work on Progger 2.0, with the initial focus being the construction of a heat map based on information extracted from the JastAdd tracing system as related in Section 4.3.

The "elastic band" component of the design has yet to manifest, as during the process of implementation and discussion around the new version of the prototype, we found ourselves returning to questions centred around the conversational framing of the tool. We know that the place in which a person looks is indicative of them paying *attention* to whatever is in that location. Therefore, if we take the lines of code that the *compiler* looks at as a sign of where its attention lies, can we use this information re-focus the *programmer's* attention, and is this alignment of attention useful in the debugging process? To answer these questions, we paused development of Progger 2.0 with only the underlying heat map technology complete, visualised with highlighting that becomes darker with a greater number of hits and shown in Figure 10, and embarked upon a new study.

## 4.6 Eye-tracking Study

Having updated Progger to include heat map information, we made the decision to test our theories about attention by way of an eye-tracking study. Eye-tracking has been used many times in the context of programming and program comprehension [CS90][Fei19], however a recent study by Busjahn et al. [Bus+15] drew our attention due its comparison of novice and expert gaze behaviour. They found that when presented with unfamiliar code, novices tend to read it in a much more linear fashion, scanning the code line by line, whilst in contrast experts were found to read non-linearly, with their gaze jumping around to different non-consecutive sections of code.

This line of research synchronised well with our previous work on Progger, as the tool was intended to aid novice programmers to debug code more efficiently. For this reason, we decided to incorporate a similar analysis of linearity into our work, albeit without the direct comparison to expert behaviour. To accomplish this,



**Figure 10:** An example of a heat map generated on a one of the Java code snippet that was presented in the user study. As the mouse pointer is hovering over the first error ("local variable count is not assigned before use" on line 14) the attention of the compiler when finding that error is shown is visualized as a line-based heat map.

we would conduct a study once again focusing on novice programmers, with the difference in their gaze behaviour between highlighted and non-highlighted versions of code analysed. To accomplish this, we again chose to recruit participants from the Lund University student body, although the hardware required to collect accurate data meant that we were unable to use the simple café study setup explained in Section 3. We therefore undertook a more active recruitment approach, advertising to students taking a computer science class on agile methodology as well as to the wider student body via Facebook groups.

As the study procedure (more details of which can be found below) would involve a closer reading of programs than in the previous user study, therefore necessitating at least a basic understanding of programming concepts, we made the completion of at least one programming class a requirement for participation. Timings were also more restrictive due to the necessity of asking participants to come in person to the Lund University Humanities Lab at a specific time slot for data collection, so to encourage participation we offered compensation for time spent



by way of a gift card for a Swedish cinema chain. In all, 15 students participated in the study.

### Study Design

To conduct the study, we created an experimental procedure using the Tobii Pro Lab eye-tracking software [Tob23]. This procedure consisted of an initial calibration step (common in eye-tracking studies, and used to ensure that eye-tracking data is accurate), followed by the presentation of eight stimuli. Each stimulus was a screenshot of a small (between 8 and 17 lines) Java program containing a single compiler error, presented within Proggy 2.0. Four of the stimuli had a basic version of the code presented - with just the error line highlighted and error message shown in the sidebar - and the other four with the addition of heatmap highlighting. Participants were asked to try to understand the code in each stimulus and to determine why the error occurred and how it might be fixed.



**Figure 11:** The experimental setup. The apparatus is contained within a booth, with the eye-tracker visible below the screen and a chin- and forehead rest in the foreground.

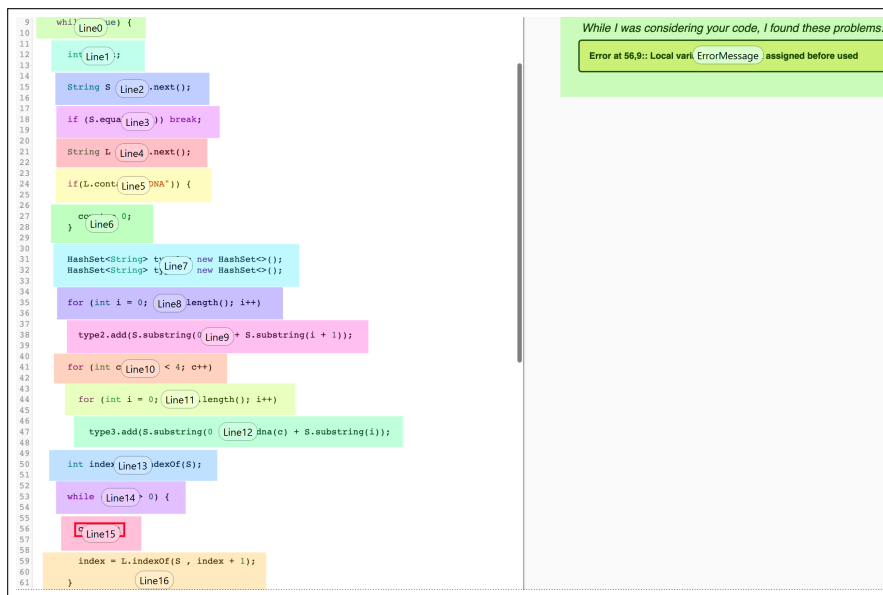
After they felt like they were able to answer these questions, participants were asked to press a key that would take them to a text-input page, where they could answer these questions in their own words. The order of stimuli was randomised per participant, and two stimuli sets were created: Set A containing stimuli 1 to 4 showing a basic code view and stimuli 5 to 8 showing the heat map view, and Set B containing the inverse of this. A picture of the setup can be seen in Figure 11. A side-by-side comparison of a basic vs. heat map highlighted stimulus is not presented due to space concerns, however they resembled the two different code views presented in Figure

10. For a direct comparison of a stimulus in its basic and heat map highlighted forms, please refer to **Paper 4**.

### Data Analysis

For the analysis of the data collected during the study, we made use of several features of Tobii Pro Lab. Most significantly, Tobii Pro Lab provides the ability to define parts of a stimulus as "areas-of-interest" (AOIs). In the context of our study, we deemed every individual line of code to be an AOI, with a screenshot showing this within the software shown in Figure 12. This allowed the recordings to be analysed within Tobii Pro Lab such that every gaze fixation (when the eye rests on

a specific point) was categorised as falling either within an AOI or without, and if falling within an AOI we were able to specifically relate that AOI to the relevant line of code.



**Figure 12:** The areas-of-interest for a stimulus, as defined in Tobii Pro Lab.

We then exported the raw data with AOI information to a tab separated values (TSV) file, a text-based representation of a table, and constructed a set of Java programs to perform additional analysis on the data. This involved defining a subset of AOIs that corresponded to lines which would be highlighted by Progger, so that comparisons could be made between highlighted and non-highlighted stimulus versions. The variables that were analysed in this way are as follows:

- **Time to completion:** the time taken to solve a given stimulus task.
- **Correctness:** a binary grading of whether the suggested solution was correct or incorrect. This was carried out as an iterative process between authors in order to ensure that we reached an agreement over the grading of each answer.
- **Hit-rate:** the number of AOI fixations calculated as a percentage on lines that would be highlighted by Progger 2.0, *regardless of what version of the stimulus was shown*. This allowed us to assess whether the highlighting functionality drew the users attention to lines that the compiler focused on,

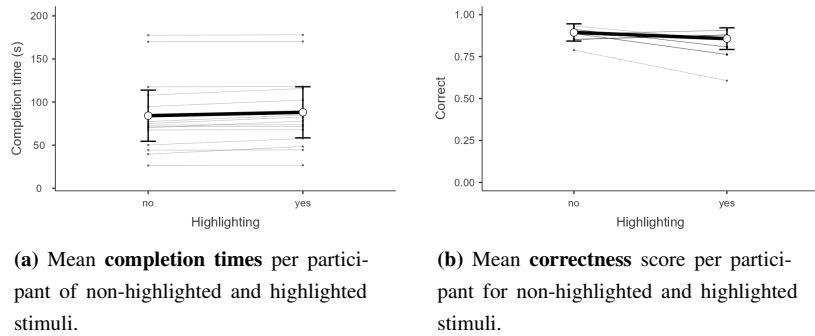
by comparing user attention on these lines between basic and highlighted states.

- **Dwell duration:** the average length of time that participants fixated on an AOI before looking somewhere else. As with **hit-rate**, this was calculated only for the sub-set of highlighted AOIs in order to determine whether highlighting affected how *length of attention* was affected by highlighting.
- **Saccade length:** the average distance between gaze fixations.
- **Linearity:** the percentage of eye movements that could be considered a linear forward change. This was achieved by analysing pairs of fixations before and after saccades, and determining whether this amounted to moving from one line of code to the next immediate line.

## Results

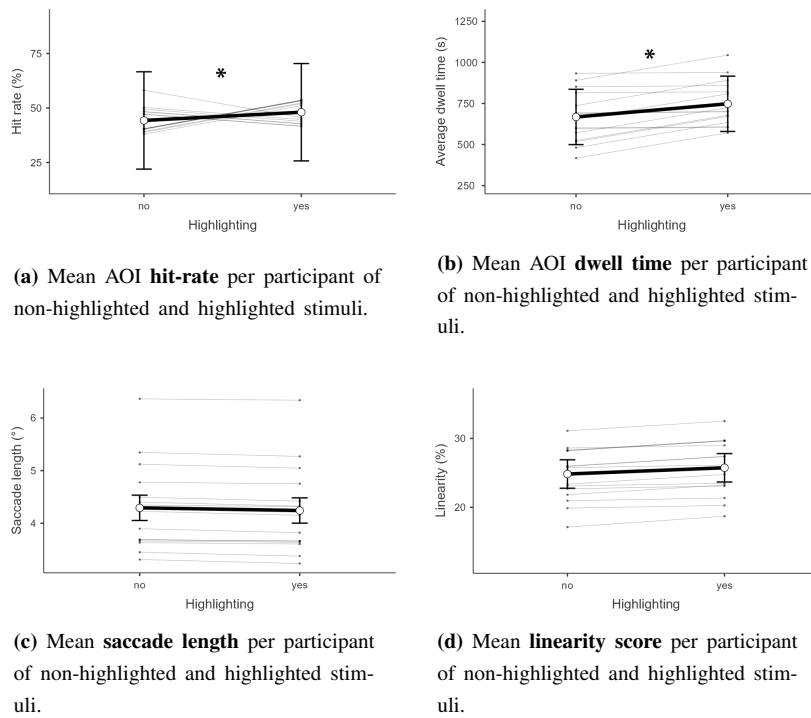
A detailed breakdown of the results of data analysis, including a description of the methods used for analysis and specific  $p$  values, can be found in **Paper 4**. However, in the interest of brevity only a high-level overview of main findings are presented here. In each of the following plots, data for individual participants is shown by thin lines, while the mean across all participants is shown by a single thick line. Standard errors of the mean are indicated by error bars.

Of the 6 dependent variables specified, the first 2, completion time and correctness, can be categorised as "performance metrics". The remaining 4 variables fall into the category of "gaze metrics".



**Figure 13:** Charts detailing the completion time and correctness metrics.

As can be seen in Figure 13(a) and Figure 13(b) respectively, neither **completion times** nor **correctness** were significantly affected by toggling highlighting on or off.



**Figure 14:** Charts detailing the collected eye-tracking data. An asterisk (\*) indicates a statistically significant effect of highlighting change.

Similarly to the performance metrics, **saccade length** (Figure 14(c)) and **linearity score** (Figure 14(d)) were not significantly affected by the highlighting state. **Hit-rate** (Figure 14(a)) and **dwell time** (Figure 14(b)), by contrast, saw significant statistical effects as a result of the highlighting, with  $p$  values of 0.002 and 0.036 respectively.

## Discussion

From the data gathered, we are able to conclude that highlighting does not have a significant effect on performance when undertaking a program comprehension task, nor does it affect reading linearity or the related metric of saccade length (where a short saccade is indicative of the eyes moving between contiguous locations). This suggests that heat map highlighting is an ineffective method for influencing the gaze patterns of novice programmers to make them more consistent with those of expert programmers. The validity of these findings may have been

influenced by the low number of participants leading to a small overall sample size. The context may also have been a factor, as the participants were instructed to attempt to understand the stimulus code with no consideration for time. This, along with the broader context of program comprehension, may have encouraged subjects to adopt a more methodical reading approach than if they were under time constraints, resulting in their scanning the text line-by-line.

In contrast to this, the findings related to the development of joint attention were much more interesting. We found that by adding highlighting to different sections of the code, novice programmers:

- Look at the highlighted sections more frequently.
- Look at the highlighted sections for longer.

This is consistent with the behaviour of the compiler: highlighted sections represent areas that the compiler looked at more frequently, for a greater amount of compilation time. This suggests that the use of highlighting within the interface design is indeed an effective way of making the user aware of the compiler's focus, and ultimately fostering joint attention.

## 5 Conclusions and Future Work

In this thesis, we have described the development of a design paradigm for programming tools centred around the mechanics of human conversation. Drawing from research into the mechanics of human conversations, we have selected specific aspects, namely *breakdowns* and *gaze as a side channel*, for exploration in the context of programming tools. With an initial focus on *breakdowns*, we applied this perspective to programming interactions in an effort glean insight into why compiler errors, specifically, often prove to be confusing and frustrating, particularly from novice users. From this analysis, we developed the Progger prototype, a tool which attempted to draw from otherwise hidden details of the compilation process in order to externalise the "thought process" of the compiler, in much the same way that a human could more precisely explain their reasoning when confronted with misunderstanding from a conversational partner.

Using the initial implementation of Progger, we then put to test our design assumptions by way of a lean user study methodology, the café study. Despite the low-cost nature of this study, it proved to reveal broad insights into the design of Progger that allowed us to strip out ineffective elements and re-focus on aspects considered to be of significant interest to study participants. This allowed for a rapid design iteration, effectively achieving the goals for the study and proving the method to be effective when searching for coarse grained sentiment feedback.

Following the design iteration occasioned by the café study, Progger 2.0 was implemented with a particular focus on code locality and highlighting through the compiler heat map. This work continued the exploration of conversational design: it constituted a refinement of the method for solving *breakdowns* via the approach of *gaze* and *joint attention* alignment. A further study resulted from this work, this time making use of eye-tracking technology. This study revealed that, though compiler heat map highlighting did not have an effect on performance metrics in a code comprehension and bug finding task, it was an effective means of directing the programmer's gaze. In this way, it appears to be an effective means of encouraging the user to direct their *attention* to code sections where the compilers attention was focused.

Our attempts to humanise programming tools by conversational design (RQ<sub>1</sub>) have led to interesting results, however the over-arching research question is yet to be fully addressed. At this point, it is unclear how "beneficial" this method of design is to users as it appears to have limited effect on performance metrics as discussed in Section 4.6. This may have been affected by the study design, however, and therefore warrants further investigation. Despite this, the effectiveness of highlighting as a means to foster joint attention has shown promising results.

For future work, we intend to further investigate the ways in which the conversational lens may be applied to interaction design in the domain of programming tools. One key feature of conversations that we have not fully explored is the idea that meaning is constructed *intersubjectively* [Sea96] - that is, all conversation par-

ticipants take an active role in establishing *conversational alignment*. In our work thus far, the computer component has taken a less active role in the conversation than the human user, with static elements computed at compile time, such as the compiler heat map, used to elaborate on its intentions.

With the recent rapid development of large language model (LLM) based AI chatbots, such as ChatGPT<sup>10</sup>, it may be possible to use these technologies to have the programming environment take a more active role in the conversation. For instance, one strategy for resolving a conversational breakdown is that of *active listening* [RF15]. This entails one participant summarising in their own words their understanding of what the other person is talking about, or repeating certain parts of the content. With use of LLM technology, it may be possible to have both the user and the programming environment state, in human language, what they believe the code should accomplish. This can help highlight divergence in understanding, and give insight into methods for repairing the breakdown.

*Side-channels* also remain a rich avenue for further exploration, as *gaze* is just one example of a side-channel. It may be of interest to investigate the use of sensors to attempt to interpret other side-channel mechanisms such as body language in an effort to gauge understanding and frustration. In general, we believe that the application of a conversational lens to programming interaction has the potential to offer many more insights into their design. By further studying this most human form of communication, we hope to continue progressing towards the humanisation of programming tools.

---

<sup>10</sup><https://chat.openai.com/>

# BIBLIOGRAPHY

---

- [ASB19] M. Ahrens, K. Schneider, and M. Busch. “Attention in Software Maintenance: An Eye Tracking Study”. In: *Proceedings of 6th International Workshop on Eye Movements in Programming (EMIP)*. 2019, pp. 2–9.
- [BB13] A. Bacchelli and C. Bird. “Expectations, Outcomes, and Challenges of Modern Code Review”. In: *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, 712–721.
- [Bas+16] A. Basman et al. “Software and How It Lives On - Embedding Live Programs in the World Around Them”. In: *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2016.
- [Bec+19] B. A. Becker et al. “Compiler error messages considered unhelpful: The landscape of text-based programming error message research”. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR)*. 2019, pp. 177–210.
- [Bed12] R. Bednarik. “Expertise-Dependent Visual Attention Strategies Develop over Time during Debugging with Multiple Code Representations”. In: *International Journal of Human-Computer Studies* 70.2 (2012), 143–155.
- [Ben+19] E. Beneteau et al. “Communication breakdowns between families and Alexa”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI ’19. 2019, pp. 1–13.



- [Bla00] A. F. Blackwell. “Dealing with new cognitive dimensions”. In: *Workshop on Cognitive Dimensions: Strengthening the Cognitive Dimensions Research Community*, University of Hertfordshire. 2000.
- [Bla02] A. F. Blackwell. “First steps in programming: A rationale for attention investment models”. In: *Proceedings of the IEEE 2002 Symposium on Human Centric Computing Languages and Environments*. IEEE. 2002, pp. 2–10.
- [Bla15] A. F. Blackwell. “Patterns of User Experience in Performance Programming”. In: *Proceedings of the First International Conference on Live Coding*. ICSRiM, University of Leeds, 2015, pp. 12–22.
- [BG00] A. F. Blackwell and T. Green. “A Cognitive Dimensions Questionnaire Optimised for Users”. In: *Proceedings of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2000.
- [Bla+18] A. Blackwell et al. “Computer Says ‘don’t Know’ - Interacting Visually with Incomplete AI Models”. Talk at DTSHPs workshop, Co-located with VLHCC. 2018.
- [BC06] V. Braun and V. Clarke. “Using thematic analysis in psychology”. In: *Qualitative Research in Psychology* 3 (2006), pp. 77–101.
- [Bus+15] T. Busjahn et al. “Eye Movements in Code Reading: Relaxing the Linear Order”. In: *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE. 2015, pp. 255–265.
- [Che+22] S. Cheng et al. “Collaborative eye tracking based code review through real-time shared gaze visualization”. In: *Frontiers of Computer Science* 16 (2022).
- [Chu18] L. Church. “Critique of ‘lector in Codice or the Role of the Reader’”. In: *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*. <Programming >. Association for Computing Machinery, 2018, p. 187.
- [CNB10] L. Church, C. Nash, and A. F. Blackwell. “Liveness in Notation Use: From Music to Programming”. In: *Proceedings of the 21st Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2010.
- [CSM21] L. Church, E. Söderberg, and A. T. McCabe. “Breaking down and making up-a lens for conversing with compilers”. In: *Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2021.

- [CS19] L. Church and E. Söderberg. “Probes and Sensors: The Design of Feedback Loops for Usability Improvements”. In: *Proceedings of the 30th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2019.
- [Cla+19] L. Clark et al. “What Makes a Good Conversation? Challenges in Designing Truly Conversational Agents”. In: *Proceedings of the 2019 Conference on Human Factors in Computing Systems (CHI)*. Association for Computing Machinery, 2019, 1–12.
- [CM12] G. Cox and A. McLean. *Speaking Code*. MIT Press, 2012.
- [CSW02] M. Crosby, J. Scholtz, and S. Wiedenbeck. “The roles beacons play in comprehension for novice and expert programmers”. In: *Proceedings of the 14th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2002.
- [CS90] M.E. Crosby and J. Stelovsky. “How do we read algorithms? A case study”. In: *Computer* 23.1 (1990), pp. 25–35.
- [Cro05] N. Cross. *Engineering design methods: strategies for product design*. John Wiley & Sons, 2005.
- [Dub04] H. Dubberly. *How do you design?* Dubberly Design Office, 2004.
- [DP09] H. Dubberly and P. Pangaro. “What is conversation? How can we design for effective conversation”. In: *Interactions Magazine* 16.4 (2009), pp. 22–28.
- [EH07a] T. Ekman and G. Hedin. “The JastAdd system—modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.
- [EH07b] T. Ekman and G. Hedin. “The jastadd extensible java compiler”. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications (OOPSLA)*. 2007, pp. 1–18.
- [Fei19] D. G. Feitelson. “Eye Tracking and Program Comprehension”. In: *Proceedings of the 2019 IEEE/ACM 6th International Workshop on Eye Movements in Programming (EMIP)*. 2019, pp. 1–1.
- [FSH20] N. Fors, E. Söderberg, and G. Hedin. “Principles and patterns of jastadd-style reference attribute grammars”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. 2020, pp. 86–100.
- [FBT07] A. Frischen, A. P. Balyiss, and S. P. Tipper. “Gaze Cueing of Attention: visual attention, social cognition, and individual differences”. In: *Psychological bulletin* 133.4 (2007), pp. 694–724.

- [Gal19] Galluci, M. *GAMLj: General analyses for linear models*. Version 2.6. 2019.
- [Gon+19] L. Gonçalves et al. “Measuring the Cognitive Load of Software Developers: A Systematic Mapping Study”. In: *Proceedings of 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 2019, pp. 42–52.
- [Gre90] T. R. G. Green. “The Cognitive Dimension of Viscosity: A Sticky Problem for HCI”. In: *Proceedings of the IFIP TC13 Third International Conference on Human-Computer Interaction*. INTERACT ’90. North-Holland Publishing Co., 1990, 79–86.
- [GP96] T. R. G. Green and M. Petre. “Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework”. In: *Journal of Visual Languages Computing* 7.2 (1996), pp. 131–174.
- [Hed00] G. Hedin. “Reference attributed grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.
- [HM03] G. Hedin and E. Magnusson. “JastAdd - An aspect-oriented compiler construction system”. In: *Science of Computer Programming* 47 (2003), pp. 37–58.
- [HH11] A. Henderson and J. Harris. “Conversational Alignment”. In: *Interactions* 18.3 (2011), 75–79.
- [Hes+18] R. S. Hessels et al. “Is the eye-movement field confused about fixations and saccades? A survey among 124 researchers”. In: *Royal Society Open Science* 5 (8 2018).
- [HE96] I. T. C. Hooge and C. J. Erkelens. “Control of fixation duration in a simple search task”. In: *Perception & Psychophysics* 58.7 (1996), pp. 969–976.
- [Imt+19] N. Imtiaz et al. “Challenges with Responding to Static Analysis Tool Alerts”. In: *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 245–249.
- [Joh+13] B. Johnson et al. “Why don’t software developers use static analysis tools to find bugs?” In: *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 672–681.
- [Kat+09] L. C. L. Kats et al. “Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing”. In: *ACM SIGPLAN Notices* 44.10 (2009), pp. 445–464.

- [Kic+97] G. Kiczales et al. “Aspect-oriented programming”. In: *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP)*. Springer. 1997, pp. 220–242.
- [Knu68] D. Knuth. “Semantics of context-free languages”. In: *Mathematical Systems Theory* 2.2 (1968), pp. 127–145.
- [Kua+23] P. Kuang et al. “Towards Gaxe-Assisted Developer Tools”. In: *Proceedings of the 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE. 2023.
- [Loh16] S. Lohmeier. “A Formal and a Cognitive Model of Anaphors in Java”. In: *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2016.
- [MEH09] E. Magnusson, T. Ekman, and G. Hedin. “Demand-driven evaluation of collection attributes”. In: *Automated Software Engineering* 16.2 (2009), pp. 291–322.
- [MSC21] A. T. McCabe, E. Söderberg, and L. Church. “Progger: Programming by Errors (Work In Progress)”. In: *Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group*. 2021.
- [MSC22] A. T. McCabe, E. Söderberg, and L. Church. “Visual Cues in Compiler Conversations”. In: *Proceedings of the 33rd Annual Workshop of the Psychology of Programming Interest Group Annual Workshop (PPIG)*. 2022.
- [MMK12] M. Mori, K. F. MacDorman, and N. Kageki. “The Uncanny Valley [From the Field]”. In: *IEEE Robotics Automation Magazine* 19.2 (2012), pp. 98–100.
- [MCB15] M. Mărăşoiu, L. Church, and A. Blackwell. “An Empirical Investigation of Code Completion Usage by Professional Software”. In: *Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group*. 2015.
- [NNQDB19] M. Nachtigall, L. Nguyen Quang Do, and E. Bodden. “Explaining Static Analysis - A Perspective”. In: *Proceedings of the 34th International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE/ACM. 2019.
- [Nie+19] D. C. Niehorster et al. “Searching with and against each other: Spatiotemporal coordination of visual search behavior in collaborative and competitive settings.” In: *Atten Percept Psychophys* 81 (2019), pp. 666–683.
- [Nor13] D. Norman. *The design of everyday things: Revised and expanded edition*. Basic books, 2013.

- [NHW96] D. Novick, B. Hansen, and K. Ward. “Coordinating turn-taking with gaze”. In: 1996, 1888–1891 vol.3.
- [OAHC18] U. Obaidallah, M. Al Haek, and P. C.-H. Cheng. “A Survey on the Usage of Eye-Tracking in Computer Programming”. In: *ACM Computing Surveys* 51.1 (2018).
- [Öqv18] J. Öqvist. “ExtendJ: extensible Java compiler”. In: *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*. 2018, pp. 234–235.
- [Oul+18] A. Oulasvirta et al. *Computational Interaction*. Mar. 2018, pp. 1–424.
- [Pas76] G. Pask. “Conversation Theory”. In: *Applications in Education and Epistemology* (1976).
- [Pug81] S. Pugh. “Concept selection: a method that works”. In: *Proceedings of the International conference on Engineering Design*. 1981, pp. 497–506.
- [R C21] R Core Team. *R: A Language and environment for statistical computing*. Version 4.1. 2021.
- [Rao+02] R. P. N. Rao et al. “Eye movements in iconic visual search”. In: *Vision Research* 42.11 (2002), pp. 1447–1463.
- [Ray98] K. Rayner. “Eye movements in reading and information processing: 20 years of research”. In: *Psychological Bulletin* 124.3 (1998), pp. 372–422.
- [RF15] C. R. Rogers and R. E. Farson. *Active Listening*. Martino Publishing, 2015.
- [SSJ74] H. Sacks, E. Schegloff, and G. Jefferson. “A Simple Systematic for the Organisation of Turn Taking in Conversation”. In: *Language* 50 (Dec. 1974), pp. 696–735.
- [Sad+15] C. Sadowski et al. “Tricorder: Building a program analysis ecosystem”. In: *Proceedings of IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 598–608.
- [Sad+18] C. Sadowski et al. “Modern Code Review: A Case Study at Google”. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’18. Association for Computing Machinery, 2018, 181–190.
- [Sea96] J. Searle. *The Construction of Social Reality*. Penguin, 1996.
- [SSG15] Z. Sharafi, Z. Soh, and Y.-G. Guéhéneuc. “A systematic literature review on the usage of eye-tracking in software engineering”. In: *Information and Software Technology* 67 (2015), pp. 79–107.

- [SH10] E. Söderberg and G. Hedin. “Automated selective caching for reference attribute grammars”. In: *Proceedings of the 3rd International Conference on Software Language Engineering (SLE)*. Springer. 2010, pp. 2–21.
- [SH12] E. Söderberg and G. Hedin. *Incremental evaluation of reference attribute grammars using dynamic dependency tracking*. Department of Computer Science, Lund University, 2012.
- [Sto05] M.-A. Storey. “Theories, methods and tools in program comprehension: past, present and future”. In: *Proceedings of the 13th International Workshop on Program Comprehension (IWPC)*. 2005, pp. 181–191.
- [Tan90] S. L. Tanimoto. “VIVA: A Visual Language for Image Processing”. In: *Journal of Visual Language and Computing* 1.2 (1990), 127–139.
- [Tob23] Tobii AB. *Tobii Pro Lab*. Version 1.217. 2023.
- [Wes11] M. West. “The Homeric Question Today”. In: *Proceedings of the American Philosophical Society* 155.4 (2011), pp. 383–393.
- [pro22] the jamovi project. *jamovi*. Version 2.3. 2022.



---

## **INCLUDED PAPERS**

---





# BREAKING DOWN AND MAKING UP - A LENS FOR CONVERSING WITH COMPILERS

---

## Abstract

This paper proposes a ‘tool for thinking with’<sup>1</sup>: that we can describe the interaction between people and computers, and especially people and developer tools, as a form of conversation. We outline this perspective, construct a work in progress analytical frame, and use it to talk about a couple of different examples and draw implications for future work.

## 1 Introduction

Most professional software engineering is currently done around a textual representation of a program in one or more languages. This representation, *the code itself*, acts as the central focus of a number of interactions. It’s typed into IDEs and text editors, reviewed in version management tools, copied and pasted as snippets into collaboration tools and online forums, sworn at with colleagues, and applauded when it works. It’s the focus of cultural events, legal disputes, and Hollywood fantasies.

In other words, code is significant, both in the professional practice of software engineers and more generally. The social status of code, and its performative role has been the centre of increasing writing in recent years [CM12], whilst our

---

<sup>1</sup> A phrase coined by Steven Clarke at the industry panel, PPIG 2016, Cambridge.

[Sad+18] and others' previous work [BB13], looking at the role that conversations around code in review tools shows a complex set of social processes, including education, status signalling, and cultural norm building. But despite all of this, the interactions with typical contemporary developer tools in use are rather limited (edit, compile, debug) and originate from tools (compiler, static analyser, virtual machine) having fairly fixed and historic roles. Compared to other software where the interaction patterns have evolved considerably these are starting to look increasingly anachronistic.

A developer might write some code, and when they press the compile button - as many build infrastructures still make them do - they get back an error. If they don't understand the error then all of the burden of what to do now falls on them and their social network to fix. If they try again, they'll just get the same error again, a particularly stubborn interaction of which one side not only can't change its answer, but won't even provide any more information. Previously, we used either the metaphor of the friend who won't read your book because of the missing full stop on page 237 [Chu18], or the computer that just says 'no' (or don't know) [Bla+18], to describe this kind of interaction.

What these show is a conversational dynamic, or rather a lack of one, between the developer and the programming infrastructure they use. The strict, pedantic, and fixed nature of this interaction contributes to a catalogue of problems, from usability issues with static analysis [Joh+13], to problems in CS Education [Bec+19], but it may also have a broader effect. It localises all of the challenges of the interaction with the formal system of code, into the code itself. This very much centres the interactions onto the terms of the computer, not the people doing the development.

This paper describes a work in progress at more closely describing this conversational dynamic, starting off with a description of some of the existing work on the analysis of conversations, building that into the beginnings of an analytical tool, applying the tool first to the description of interaction in general, then to interaction with a compiler, and finally as a motivation for building an experimental platform.

## 2 Aspects of Conversations

In this section, we list an initial selection of aspects of conversations which we later use as an analytical perspective to describe the exploration of a conversational form of interaction with tools like compilers<sup>2</sup>. Whilst the notion of considering interaction as a conversation has been considered earlier by, for instance, Hugh

---

<sup>2</sup>For convenience we'll refer to this as 'conversations with a compiler', this is expanding the role of the compiler to be the technology that handles all the underlying information structure behind an IDE offering services such as code completion, and the build process - there wasn't a particularly good name for all those things, so we'll use compiler in the broadest possible sense of programming language interaction, tooling and infrastructure.

Dubberly and Paul Pangaro [DP09] building on the work of Gordon Pask [Pas76], it has not to our knowledge been further explored in the context of programming tools.

Our starting point for this exploration is an informal selection of previous work related to conversations; conversation theory [Pas76], conversation analysis [SSJ74], interaction design and conversations [DP09], conversational alignment [HH11], communication breakdown [Ben+19] and properties of good conversations [Cla+19], we consider the following groups of aspects: (1) “Turns & Temporality”, (2) “Intersubjectivity, Alignment & Active Listening”, (3) “Tolerance, Breakdown & Repair”, (4) “Explicability”, and (5) “Side-channels & Deixis”. We will describe these aspects here in terms of normal person-to-person conversations and connect them to the selected literature and other work describing programming language tooling.

## 2.1 Turns & Temporality

Conversations have a cadence to them, often one person speaks and leaves pauses for the other person to speak. If they want to, the other person then starts speaking and takes ‘their turn’. If people want to interrupt they will often signal this implicitly, or start to speak if there is a gap and back off if the other person isn’t done. In some cases (small children, large conferences) this logic is explicitly supported. Sometimes by having a physical token (‘you can talk when you have the ball’), other times by structured ‘question and answer’ times. In closer conversations between friends, the conversational structure can become a little more informal with people taking over conversations midway through sentences.

Dubberly & Pangaro [DP09] describe the structure of a conversation as a process where participants open a channel, commit to engage, construct meaning, evolve, converge on agreement, then act or transact. A central aspect of this process is turn-taking, described in a model by Sacks et al. [SSJ74]. This turn-taking model includes turn-constructive components (how a speaker constructs a turn), turn-allocation components (how to allocate who gets the next turn), and rules, such as if the current speaker selects the next speaker, then the selected speaker is obliged to take the next turn to speak.

The closest analogy to turn taking that we are aware of in the study of programming tools is the descriptions of liveness and the temporal nature of interaction within the live programming community. Tanimoto’s framework [Tan90], and the broader live programming communities have studied the intertemporal nature of the interaction between developers and their tools about the code, and whether these are primarily episodic (traditional build compile cycles) in nature or more continuous such as the example above, and whether these timing characteristics are for interaction with the code, or with the running program.

While our reading of the models of Pask and Sacks et al. is at an early stage, we note that they have different origins. The conversational model by Pask stems from

cybernetics and is not directly bound to human-to-human conversations, while the turn-taking model by Sacks et al. originates from studies of human conversation. In relation to the intersection of these two models, a recent study by Clark et al. presents a difference in expectations on a human-to-human conversation and human-to-agent conversion, comparing the latter with that of a conversation with a stranger [Cla+19]. How these models overlap has interesting applications to the domain of interacting with compilers but at this point represents future work that's beyond the scope of this paper.

## 2.2 Meaning Making: Intersubjectivity, Alignment & Active Listening

The construction of meaning within a conversation is obviously a complicated topic of epistemology, and we can only present a very preliminary and high-level way of thinking about it here, focussed on the end of understanding conversations with computers.

Meaning is sometimes described as being constructed *intersubjectively* [Sea96], that is between the people in the conversation, and that a number of mechanisms are used to determine *conversational alignment* [HH11] - whether they are "on the same page". Utilization of conversations to create a shared understanding ("*meaning making*") and to reach agreement [DP09] is central in conversation theory [Pas76], where conversations are seen as interactions between cognitive processes and as key drivers for learning, as different models of understanding are reduced to a shared model. In a recent empirical study by Clark et al., mutual understanding was found to be one aspect of a good conversation, alongside trustworthiness, active listening, and humour [Cla+19].

This implies that the conversation might not be meaningfully interpretable outside the context of the conversation, for example, when reading what was written after a substantial period of time has passed, or if others that weren't part of the original conversation read it. For example a reference to shared experiences such as 'where did I put the thing that we brought back from that holiday in the mountains?' might make complete sense to your friend, but wouldn't mean anything to someone else. Some techniques for having conversations elevate these practices from instinctive to intentional habits. For example, in *active listening* [RF15] one of the techniques used to ensure alignment between participants is for one of them to summarise the content of what is being said, or repeat what they heard. This gives the other participants a chance to see whether they're 'getting it'.

## 2.3 Tolerance, Breakdown & Repair

As the meaning is intersubjective in the conversation between two people it's inevitable that their understanding won't be exactly the same. To keep the conversation going, we suspend trying to build a precise shared understanding until it really

matters. We may, for instance, tolerate that we don't have a definition for some of the terms, or a precise description of what they do or don't include.

Sometimes however, it becomes clear that the misunderstanding is significant enough that you're actually talking about completely different things, at which point the conversation 'breaks down'. At a point of this *communication breakdown*, the normal flow of the conversation stops and instead either the conversation ends, somewhat acrimoniously, or, more commonly, the participants in the conversation attempt to *repair* it [SSJ74]. In this repair, they enter what may be referred to as a meta-conversation [DP09], where the participants attempt to establish what the source of the misunderstanding is, clarify, and then go back and proceed with the conversation.

In a recent study by Beneteau, et al. [Ben+19], studying human-to-agent conversation by observing how families interact with the conversational agent Alexa, they found that the burden of the repair was primarily on the humans. Alexa could signal a breakdown (e.g., "did you mean X" or "sorry I'm not sure"), but provides next to no assistance with repair (e.g., could indicate a need for assistance with a definition). The participants in the study used several repair strategies, e.g., adjusting their cadence to that of the agent, exaggerating sounds (hyperarticulation), adjusting sentence structure to clarify (e.g. from "alexa, thank you, stop" to "alexa, stop"), and repeating the previous utterance again.

In relation to programming tools and compilers, in a study by Johnston et al. on why software developers don't use static analysis tools [Joh+13] they found usability issues connected to false positives, workflow integration, overflow of results, and comprehensibility of results. In a related study by Imtiaz et al. in analyzing questions about static analysis tools on the popular StackOverflow platform [Imt+19] found the most common question to be about how to ignore results. With the lens of conversations, several of the found usability issues with static analysis results can be considered as breakdowns. Again, the primary burden of the repair is on the human and based on the common practice of ignoring results there is not much of a conversation. In Basman et al. [Bas+16] we have discussed the various technical sources from which this breakdown will occur, but primarily from the perspective of structurally avoiding them rather than building mechanisms through which they can be repaired.

## 2.4 Explicability

The repair mechanism outlined above is a form of *explicability* - where one side (if possible) asks for more details or more description on a phenomena that has occurred. This explicability may be guided, where one party asks questions in order to shape the information they are seeking (or, in the case of a Socratic dialogue, encouraging self-reflection about), or it may be a description that one of the participants of the conversation leads. The explanation does not have to be a repeat of the information. It might be achieved by trying to say the same thing in a

different way, providing a different example of the same thing, or analogy between the object being described and another item. This may then be coupled with active listening techniques where they try and describe what they have just heard to see if they have now understood.

As a form of repair [SSJ74], explicability is closely related to the construction of a shared understanding [Pas76] and meaning making [DP09], where it helps to bring about conversational alignment [HH11].

Explicability has recently gained prominence in Software Engineering through the drive to create ‘explainable Artificial Intelligence’, that is statistical systems that are legible in the processes they used to make decisions. Nachtigall et al. [NNQDB19] apply a similar terminology for characterising interaction with static analysis, listing a number of explainability challenges incorporating usability challenges such as incomprehensible messages, workflow integration, and false positives, also reported in, for instance, Johnson et al. [Joh+13].

## 2.5 Side-channels & Deixis

So far the description above has been focussed on the linguistic content of the channel. However this is by no means sufficient as a description of the phenomena of a conversation. There are many other things happening in the conversation; participants will be observing each others’ body language, facial expressions, tone of voice, and cadence of speech. All of these are used to give cues as to whether the conversation is making sense, whether it contains too much information or too little, whether it’s an enjoyable discussion or whether it’s frustrating.

These *side-channels* vary in different conversational settings. In one-to-one conversation you might notice your conversation partner glancing at the clock as an indication that the discussion might need to wrap up soon, while in a conference setting this more likely is signalled by the participants reading their email. As well as providing meta cues about the conversation, these channels can also be used to directly provide information, such as deictic pointing at an object and saying ‘let’s put the book on the shelf over there’, or to direct turn-taking in a conversation [NHW96].

We aren’t aware of a significant literature applying communicative side channels within programming tools, as we’ll see later, without a larger conversational frame it’s hard to know how the information gained via a side channel would be used by the tool. There have been some experiments introducing anaphora into existing programming languages [Loh16], however these remain largely experimental.

## 3 Frame: Interaction as a Conversation

Having now outlined the overall view of the conversational approach we will take, we will now apply this to describing a general interaction design problem before

using it to describe the interaction with a compiler. In the time honoured tradition of PPIG, we will describe a microwave oven. Following the lessons of operationalising the Cognitive Dimensions using a questionnaire [BG00] we performed this description by asking a series of questions about the context and each of the properties. We list these questions in Appendix 1. As with other analytical perspectives that are used to describe the interaction with programming such as Cognitive Dimensions [GP96] and The Patterns of User eXperience [Bla15], it is important to also describe the context in which the interaction is occurring.

**Context:** In the case of the microwave interaction, the conversation is between a hungry person and a microwave. The conversation is happening in the person's kitchen at eye level where the microwave is mounted on a wall. They are having the conversation because the first author would like some warm soup. The language they are speaking in is wattage and time in minutes. Now we can consider the interaction in terms of the properties we described earlier.

**Temporality:** The interaction is initiated by the person pressing the power button, at which point the microwave responds by suggesting how long it's going to cook for. It's then the person's turn to twirl a dial and press start.

At which point the microwave will begin its cooking until it's done, it will then signal that it's turn is over with a loud beeping noise. This will continue from time to time until the person acknowledges it by opening the door and closing it again. Interruptions are a one way flow with this model of microwave, if the person opens the door, cooking stops straight away. Resuming the 'microwave's turn' is an explicit action - closing the door and pressing the start button. On the other hand, whilst the person can interrupt the microwave at any point, the microwave does not interrupt, it does the same thing until it's completed its turn and then waits - possibly forever.

**Meaning Making:** The interaction is held on pretty fixed terms, four separate power levels (60, 360, 600, 1000) and the time. Whilst the person using the microwave may not be able to assign meaning to these beyond (a little amount of heating, not much heating, a fair amount of heating and a lot of heating), there is no notational change occurring on the microwave side and no tolerance of any variation from the set pattern of interaction. If it is not followed, nothing will happen. In this sense whilst there is some intersubjectivity, the person does all the learning, and if a piece of metal is introduced into the microwave there might be another opportunity for learning.

**Breakdown & Repair:** As suggested above, there are various things that the person can do to interrupt the normal usage of the microwave, for example opening the door. This will cause the microwave to stop everything it is doing, and periodically make alarm noises until the door is closed and the start button is pressed. This is the one and only way in which the conversation can be repaired, and is explicitly signalled on the user interface of the microwave.

**Explicability:** The microwave is fairly inscrutable. Whilst there is a display that explains the state (cooking, cooling, waiting for the door to be closed) there



isn't any way of requesting more information, from the significant - "why did sparks come out when I cooked my fork?" to the more mundane "how long have you been cooking for?". The former might only be discoverable by reading an encyclopedia, the latter is just a feature that isn't implemented though it of course could be with relative ease. The microwave also never requests more information from the person.

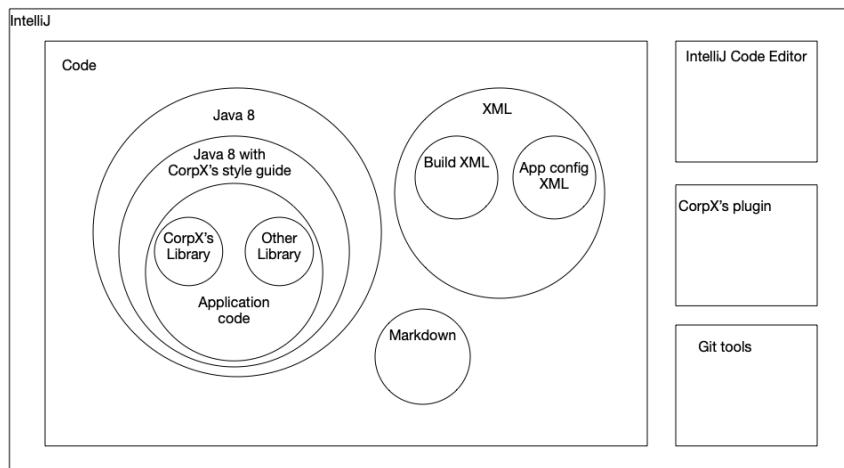
**Side-channels:** However whilst the conversation isn't subject to any form of direct elaboration, it's rich in side channels. When the microwave is running it makes a deep rumbling noise, vibrates slightly and illuminates the compartment. Over time you can see the food start to boil, and if ignored long enough this will be coupled with an olfactory side-channel as well. There are no side channels by which information can flow from the person to the microwave, it is ignorant of the world it sits in, and just performs the same series of actions in response to the same series of input, independently of happiness, hunger, or impatience waiting for the soup.

**What is all this telling us?** This description has shown what a conversation with a relatively fixed appliance looks like, and how even with a very simple device the lens of interaction turn taking, interruptions and repairs and the richness and characterisation of the side channels is an informative description of the interaction and highlights possibilities for improving explicability. We will now apply the same lens to describe interactions with a compiler.

## 4 Conversations with Compilers

Having now seen what it looks like to think of using a microwave as a conversation, we can now move on to considering a compiler. Just as there is variation in the context of use of an oven (using a domestic microwave to cook soup is different from using an industrial autoclave to cook a spacecraft fuel tank), there is also variation in the nature of software being written. In order to consider the conversation we need to be specific about the context of interaction, not just the technologies involved.

**Context:** For the purpose of this conversation, we'll try and describe a circumstance that is specific but likely to be representative of a number of activities that software engineers in the wild do. The conversation is between a software engineer and their tooling around an application written in Java. The conversation is mainly happening in an IDE such as IntelliJ or Visual Studio. They are having this conversation as the engineer has been tasked with adding another feature to the application, in a hurry but not a desperate one. The conversation is happening in multiple languages, firstly and most obviously in the Java programming language - but the story of the use of language in an IDE is complicated. Even depicted simplistically as in Figure 1, there are a lot of languages involved.



**Figure 1:** A schematic outline on the number of different languages, both notational and interactional that are involved in a notional commercial software development context (called CorpX).

Though we said that the program was ‘written in Java’, as might be a common statement in a discussion on hiring, it’s really more complicated than that. Whereas Java imposes a standard syntax and how it is interpreted<sup>3</sup>, organisations often decide to make use of only a subset of the possibilities through using tools like corporate style guides, producing a dialect of Java. The IDE will have a particular way of rendering the code with syntax highlight and font etc. For any software of any size within that dialect, a specialist vocabulary or jargon for the purpose of the software is constructed. For a furniture maker this might be the types of panels they are using to construct their pieces, for music software it might be about the acoustics of various instruments. Whilst these are both written in the grammar and syntax of Java, the libraries that express them rapidly form their own little language that isn’t easily understood. Anecdotal evidence suggests that more of the work of onboarding a new software engineer into an organisation is taken up learning these ‘little’ languages, than learning the ‘big’ language.

Apart from the ‘primary’ language, there are a whole host of peripheral languages involved. These include configuration languages: both the package management and build system, which will be fairly conventionalised between organisations and contexts, and the configuration, that will be application specific, as well languages for documentation, automation and other process support. Aside from these obviously linguistic artifacts, there are many other elements, including

<sup>3</sup>Even this turns out not to be true often, as organisations and frameworks build code transformation tools that mean that the code that appears in the editor is not the same as the code that is actually executed.

the language used on the buttons and interactive elements of the IDE, the language used in organisation specific plugins and tools like static analysers, and the language used to interact with other tools, such as unit testers (green circles meaning good), and elements of the build process such as version control (the commands used to make Git do things)

This complex linguistic environment has a number of effects, it can be fairly overwhelming for new speakers. It also creates a very viscous [Gre90] ecosystem where any improvement has to be supported across a number of different tools in order to achieve practical usability within an organisation. This partially contributes to why the infrastructure for professional development so significantly lags behind research prototypes for improved programmer experience.

As would be expected for such a complex environment, there are many aspects of temporality to consider, with the different notations having different levels of liveness [CNB10]. For the purpose of this discussion and motivating our subsequent experiment, we will focus on the conversation between the developer and their code in the primary code view, including syntax highlighting and the presentation of any errors and warnings that occur.

**Temporality:** The cadence of the conversation is primarily led by the developer who makes a change to the code. In some cases the editor responds pretty much in between keystrokes, for example updating syntax highlighting. In other cases the compiler waits for a short pause where the developer is no longer typing and does the more arduous work of computing errors etc. However once that process has started they are just blurted as soon as they are ready, potentially interrupting the flow of a further conversation that's started. So the developer's activity is partially used as a way of signalling when it would be a good time for the compiler to do something.

It is not necessarily the case that systems that are more live are better, for example some editors insert keystrokes on behalf of the developer, such as closing quotes for them. This results in the developer needing to enter into a closed loop interaction, monitoring what the editor is doing for potential incorrect interruptions that need to be fixed up, a known design flaw in adaptive text entry systems [Oul+18].

The processes that take more time on the other hand are often explicitly signalled. The developer presses a button that begins a compilation process. At which point the compiler infrastructure does its work pretty much regardless of any further input, apart from an explicit instruction to cancel, and returns the results to the developer when it's done.

**Meaning Making:** Meaning making in programming systems is a topic of considerable historical focus of the programming language community, and beyond. We suggest a part of the interaction that has a strong conversational aspect is code completion. If the developer introduces a new method successfully ('here's an idea - musical instruments can be played'), in subsequent interactions the compiler will refer back to that 'idea' ('if you're talking about an instrument, would

you like to play it?’). If on the other hand the compiler didn’t understand the method, for example if it had unbalanced braces, then this suggestion won’t occur.

Likewise when new elements are added to the program, such as methods or classes, these often appear in an adjacent area of the display. This can be thought of as another form of ‘active listening’ where the tool confirms that it has correctly understood some of the intention of the programmer by showing where in the structure of the program they have entered the new element.

We suggest that one of the conversational properties that the useful awkwardness [Bla00] of strong type systems brings, is that it is easier to support the meaning-making properties of the conversation by allowing the tooling to more completely model the program without executing it.

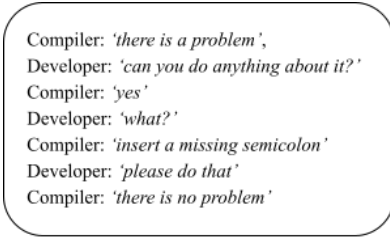
**Tolerance, Breakdown & Repair:** Different aspects of the conversation with a compiler have different levels of tolerance to mistakes. As we suggested above, many aspects are highly intolerant to the slight syntactic slips that occur frequently in day-to-day conversations between people, refusing to do any significant work with code before it is in a grammatically perfect state.

However whilst this is the case for the conversations about compilation, the other conversations that happen have wider variation in their degrees of tolerance. For example, syntax highlighting one function typically wouldn’t be prevented by another function containing a mistake like a missing semicolon, however such a mistake would stop compilation happening.

As would be expected in a situation where there are a number of different levels of tolerance, there are also different ways of signalling that the conversation has broken down, and different ways of repairing it. One example is listed above, where the divergence [Bas+16] between the developers expectation as to the elements that are available to the program and the compiler’s model is revealed by the code completion mechanism. Experienced developers use the change in behaviour that code completion is no longer suggesting ‘the right things’ as an indication that there is a problem in the code [MCB15], and look to fix the issue determining whether it has been fixed by whether code completion starts working properly again or not. This is an example where sensitivity to their alignment with their compiler appears to be an indicator of expertise.

Another example of the implicit signalling of the breakdown of alignment between the developer and the compiler occurs when the syntax highlighting goes awry. For example in the case where the developer has forgotten to close a string literal quote, suddenly all the text in front of them changes colour, which signals that the compiler is interpreting the code differently to the developer, and for an experienced developer is often a quick fix.

In other cases however the breakdown is more explicitly communicated, for example by the compiler indicating an error. This kind of breakdown is indicated by adding red squiggles underneath the text where the compiler thinks there’s an error, displaying an error in the error list beneath the text, and changing the colour of the file.



```

Compiler: 'there is a problem',
Developer: 'can you do anything about it?'
Compiler: 'yes'
Developer: 'what?'
Compiler: 'insert a missing semicolon'
Developer: 'please do that'
Compiler: 'there is no problem'

```

**Figure 2:** A notional conversation where the compiler knows of a correction.

However, as with the repairing code completion the burden is very much on one side. The developer needs to find and fix the problems with little assistance from the compiler. Some compilers such as Dart in Visual Studio Code have ‘suggested fixes’ that they can perform, but these have to be explicitly requested, as shown in Figure 2.

As well as the burden of fixing the problems falling heavily on the developer, the tone of the indication of the breakdown is often terse compared to the way in which normal conversations would be held, with error messages such as ‘variable cannot be used before declared’. The nature of the interaction tends to be one sided with the compiler having no way of sensing whether the developer’s understanding of the model of the compiler has broken down, and no way of addressing it.

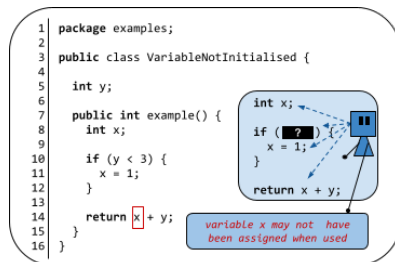
**Explicability:** Part of this lack of a way of addressing the potential breakdown between the compiler and the developer is associated with a lack of the ability to finesse the description of an error. Most compiler error messages are delivered complete to the user associated with the point in the code that they occur at. This typically adds to both the terseness described above and a barrier to the amount of conversational repair that is possible.

There is no way for a developer to ask the compiler basic questions like “why is that a problem?”, or “what were you doing when you had this problem?”, or “can you show me another example of this problem?”. These requests would be part of a normal conversation with an experienced developer in understanding why something was going wrong and what could be done about it, however the conversation with the compiler shows much more limited interactivity. It simply repeats its statement about what the problem was with no variation. This means that if the original error message didn’t help, the developer is reduced to performing trial and error to see if it changes the message that the compiler gave rather than being asked to ask for any form of refinement. This lack of explicability is the starting point for our experiment below.

**Side-channels:** Compared to the richness of the interaction with the microwave, the compiler has very limited side channels. There isn’t much indication that the compiler is doing something other than if it happens for a long time the fans start to make noise. There have been a number of attempts to use physiological data rather from eye tracking to skin salinity to observe the state of the developer but these techniques have broadly not been adopted.

## 5 Prototype: Mitigating Breakdowns in Compiler Interaction

The analysis in Section 4 paints a picture of a rather limited conversational interaction within a very complex environment. Many of the limitations in the conversational interaction occur due to the static nature of the communication with the compiler<sup>4</sup>, the developers provide code and the compiler is given the opportunity to respond, but there is no possibility of further interaction with respect to the information provided. This creates a number of problematic dynamics where the compiler stubbornly replies with the same answer as before, and offers no help, as it has no memory of the history of the conversation, or how the information that it has about the code could be used to support better explicability. How would the interaction look if we explicitly designed for breakdowns in the conversational alignment between the developer and the compiler?



**Figure 3:** Java code example with the compiler and its view illustrated with blue figures.

erties above.

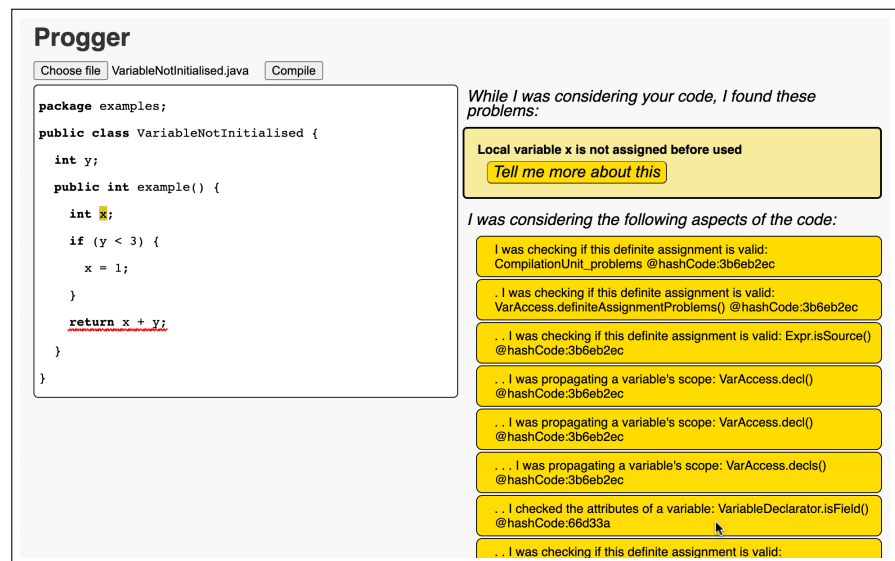
Figure 3 illustrates an example where a breakdown in the “compiler conversation” may occur. In it, a small method is used to return the sum of two variables,  $x$  and  $y$ , where  $y$  is declared as a field within the class and the value of  $x$  is set depending on the value of  $y$ . Given a default value of zero for  $y$ , we can see at a glance that the condition in the `if` statement would evaluate to true, and as a consequence  $x$  would always be assigned. The compiler however, represented by the blue robot, sees things differently - although it considers the declaration and assignment of  $x$ , it is unable to determine the result of the conditional, and therefore throws an error that may come as a surprise to the developer. To solve this

In order to experiment with this question, we built a prototype where we explore how the “compiler conversation” can continue beyond an error and the breakdown it incurs. A literal application of the conversational metaphor would result in an interaction that was similar to a conversational agent, whilst interesting as a possibility this would create a very significant implementation challenge to avoid uncanny valley effects [MMK12]. Instead we aim to implicitly support the conversational nature of the interaction outlined in the prop-

<sup>4</sup>In order to help highlight the conversational nature of the interaction with a compiler we’re going to talk as if the compiler is a person having the conversation with the developer. However this is not to imply that we think that a compiler has agency beyond the engineers that created it.

breakdown, we have built a prototype web application that attempts to act as a visualisation of the behaviour represented in Figure 4.

This prototype, named “Programming by Errors”, or “Progger” for short, consists of an extension of ExtendJ [EH07a], a compiler built upon the JastAdd meta-compilation system [HM03]. JastAdd is an implementation of the *referenced attribute grammars* formalism, as described in [Hed00], which introduces declaratively defined objects called attributes that may be attached to nodes in the abstract syntax tree and evaluated at an on-demand basis. This gives the advantage of allowing access to the attribute evaluation stack, as opposed to just the call stack, and allows us to track the evaluation of an error through different sections of the syntax tree and their corresponding tokens in the source code. With this information in hand, we present a list of errors generated by the compiler, which can be further expanded into a tree showing all of the attribute dependencies which were needed to compute each error. These attribute nodes may be hovered over to highlight the section of code that the attribute relates to in the abstract syntax tree.



**Figure 4:** Screenshot of the Progger prototype showing an expanded error view. The highlighted variable (`x`) corresponds to the code location investigated while evaluating the attribute where the cursor is hovering.

## 6 Discussion and Implications for Future Work

In this work we have sketched out an initial framework for describing conversational aspects of an interaction, and applied that to characterising how conversations with compilers currently proceed. Based on this analysis we briefly outline how an alternative might be constructed.

This alternative is not conversational in the sense of a conversational agent, but rather is conversational in terms of structural properties of the interaction, such as turn taking and explicability. Whilst it is a work in progress it shows a number of possibilities, and also highlights the technical and interaction challenges.

The primary challenge in the conversation remains bridging the different models involved. That is, between the informal models and learnt patterns of the developer and the formal representation that the compiler uses in order to compute properties of the program that it needs to compile the code. The success or failure of this bridge either allows conversational alignment mechanisms to take place, or interferes with them. The purpose that the analytical lens provides is a tool for thinking about what properties such an interaction needs and where the shortfall compared to a person-to-person conversation will create a significant opportunity for novel interaction design.

**Future Work** One possibility for future work is to look for intermediary artefacts that are well suited to tracking alignment and detecting breakdowns. This is part of the role we saw type systems and code completion playing where they acted as a mechanism by which the developer could make sure that the compiler's representation was aligned to what they had expected. There may be other properties that could be designed like this.

A second potential avenue of further research could be the one that Progger is outlining, of progressive explication of the causes of errors, led by a discussion with the programmer rather than the all-or-nothing logic to date. In building the prototype, we found the mapping of some of the activities being performed to be challenging - work to clarify the activities of the compiler in terms of the model that the developer holds of it will require further work.

In this work, we have focussed on errors, however there are many other areas of the compilers (in the broadest sense) activity that might potentially benefit from further explication, these include package and version resolution or runtime behavioural analysis.

A third potential avenue that could be pursued is to expand the richness of the communication channels that are available between the developer and the compiler, including the addition of more signals. From the compiler to the developer, this could include outlining the regions of code that have been read or changed, or the current activity or phase of the compiler. From the developer to the compiler, as our previous work on sensors and probes [CS19] suggests, it is possible to systematically approach the collection of data that shows the utility, or lack thereof,



of the interactions that the compiler is providing and that opens the possibility for supporting adaptation in the behaviour of the development environment.

## 7 Acknowledgements

This work is supported by the Swedish Foundation for Strategic Research under Grant No. FFL18-0231 and the Swedish Research Council under Grant No. 2019-05658.

## References

- [BB13] A. Bacchelli and C. Bird. “Expectations, Outcomes, and Challenges of Modern Code Review”. In: *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, 712–721.
- [Bas+16] A. Basman et al. “Software and How It Lives On - Embedding Live Programs in the World Around Them”. In: *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2016.
- [Bec+19] B. A. Becker et al. “Compiler error messages considered unhelpful: The landscape of text-based programming error message research”. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR)*. 2019, pp. 177–210.
- [Ben+19] E. Beneteau et al. “Communication breakdowns between families and Alexa”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI ’19. 2019, pp. 1–13.
- [Bla00] A. F. Blackwell. “Dealing with new cognitive dimensions”. In: *Workshop on Cognitive Dimensions: Strengthening the Cognitive Dimensions Research Community*, University of Hertfordshire. 2000.
- [Bla15] A. F. Blackwell. “Patterns of User Experience in Performance Programming”. In: *Proceedings of the First International Conference on Live Coding*. ICSRiM, University of Leeds, 2015, pp. 12–22.
- [BG00] A. F. Blackwell and T. Green. “A Cognitive Dimensions Questionnaire Optimised for Users”. In: *Proceedings of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2000.

- [Bla+18] A. Blackwell et al. “Computer Says ‘don’t Know’ - Interacting Visually with Incomplete AI Models”. Talk at DTSHPS workshop, Co-located with VLHCC. 2018.
- [Chu18] L. Church. “Critique of ‘lector in Codice or the Role of the Reader’”. In: *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*. <Programming >. Association for Computing Machinery, 2018, p. 187.
- [CNB10] L. Church, C. Nash, and A. F. Blackwell. “Liveness in Notation Use: From Music to Programming”. In: *Proceedings of the 21st Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2010.
- [CS19] L. Church and E. Söderberg. “Probes and Sensors: The Design of Feedback Loops for Usability Improvements”. In: *Proceedings of the 30th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2019.
- [Cla+19] L. Clark et al. “What Makes a Good Conversation? Challenges in Designing Truly Conversational Agents”. In: *Proceedings of the 2019 Conference on Human Factors in Computing Systems (CHI)*. Association for Computing Machinery, 2019, 1–12.
- [CM12] G. Cox and A. McLean. *Speaking Code*. MIT Press, 2012.
- [DP09] H. Dubberly and P. Pangaro. “What is conversation? How can we design for effective conversation”. In: *Interactions Magazine* 16.4 (2009), pp. 22–28.
- [EH07a] T. Ekman and G. Hedin. “The JastAdd system—modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.
- [Gre90] T. R. G. Green. “The Cognitive Dimension of Viscosity: A Sticky Problem for HCI”. In: *Proceedings of the IFIP TC13 Third International Conference on Human-Computer Interaction*. INTERACT ’90. North-Holland Publishing Co., 1990, 79–86.
- [GP96] T. R. G. Green and M. Petre. “Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework”. In: *Journal of Visual Languages Computing* 7.2 (1996), pp. 131–174.
- [Hed00] G. Hedin. “Reference attributed grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.
- [HM03] G. Hedin and E. Magnusson. “JastAdd - An aspect-oriented compiler construction system”. In: *Science of Computer Programming* 47 (2003), pp. 37–58.

- [HH11]      A. Henderson and J. Harris. “Conversational Alignment”. In: *Interactions* 18.3 (2011), 75–79.
- [Imt+19]    N. Imtiaz et al. “Challenges with Responding to Static Analysis Tool Alerts”. In: *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 245–249.
- [Joh+13]    B. Johnson et al. “Why don’t software developers use static analysis tools to find bugs?” In: *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 672–681.
- [Loh16]     S. Lohmeier. “A Formal and a Cognitive Model of Anaphors in Java”. In: *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2016.
- [MMK12]    M. Mori, K. F. MacDorman, and N. Kageki. “The Uncanny Valley [From the Field]”. In: *IEEE Robotics Automation Magazine* 19.2 (2012), pp. 98–100.
- [MCB15]    M. Mărăşoiu, L. Church, and A. Blackwell. “An Empirical Investigation of Code Completion Usage by Professional Software”. In: *Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group*. 2015.
- [NNQDB19] M. Nachtigall, L. Nguyen Quang Do, and E. Bodden. “Explaining Static Analysis - A Perspective”. In: *Proceedings of the 34th International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE/ACM. 2019.
- [NHW96]    D. Novick, B. Hansen, and K. Ward. “Coordinating turn-taking with gaze”. In: 1996, 1888 –1891 vol.3.
- [Oul+18]    A. Oulasvirta et al. *Computational Interaction*. Mar. 2018, pp. 1–424.
- [Pas76]     G. Pask. “Conversation Theory”. In: *Applications in Education and Epistemology* (1976).
- [RF15]      C. R. Rogers and R. E. Farson. *Active Listening*. Martino Publishing, 2015.
- [SSJ74]     H. Sacks, E. Schegloff, and G. Jefferson. “A Simple Systematic for the Organisation of Turn Taking in Conversation”. In: *Language* 50 (Dec. 1974), pp. 696–735.
- [Sad+18]    C. Sadowski et al. “Modern Code Review: A Case Study at Google”. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’18. Association for Computing Machinery, 2018, 181–190.

- [Sea96] J. Searle. *The Construction of Social Reality*. Penguin, 1996.
- [Tan90] S. L. Tanimoto. “VIVA: A Visual Language for Image Processing”. In: *Journal of Visual Language and Computing* 1.2 (1990), 127–139.



# PROGGER: PROGRAMMING BY ERRORS (WORK IN PROGRESS)

---

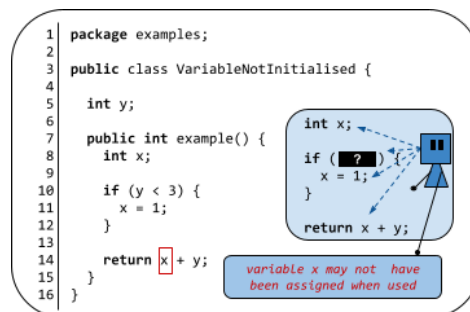
## Abstract

This paper describes a work in progress implementation of a programming tool that puts errors and their provenance at the forefront of the interaction between a developer and a compiler. We discuss the motivation for such a tool, its design and implementation, and reflect upon avenues for further research which it can facilitate.

## 1 Introduction

For novice programmers, compiler errors are a common occurrence as they work through the process of familiarising themselves with the syntax of a programming language. They write some code, execute it, and any errors that result can then be used to inform their next steps. Sometimes, however, unclear error messages can leave them at a loss as to how to proceed or, even worse, lead them down a wrong path altogether if the actual source of the error ends up being in a different location in the code from the line that the compiler flagged as problematic [Bec+19]. In this case, the respective understanding of developer and compiler are no longer in alignment [HH11] - looking at this interaction through the lens of a conversation [DP09], it can be said that a communication breakdown [Ben+19] has occurred. This conceptual approach is explored at length in [CSM21], which proposes a theoretical ‘tool for thinking with’, this paper explores a possible application of that framework.

In the event of a breakdown in understanding between two human participants, the focus would shift towards a meta-conversation about the conversation itself [DP09]. The participants would attempt to “repair” the breakdown, bringing all parties back into alignment before returning to the primary topic. A compiler, however, is much less forgiving. If a developer fails to understand a statement from the compiler, the onus is on the developer to seek out clarification and decipher exactly what is meant, with nothing from the compiler by way of assistance.



**Figure 1:** Java code example with the compiler and its view illustrated with blue figures.

As an example, if we consider the Java code in Figure 1, it has a small method (example) using two variables; one field (y) declared in the class and one local variable (x) declared in the method. The local variable is “possibly assigned” before it is used in the return statement (on line 14). If we know that the field (y) is given a default value of zero we can see that the condition in the if statement will always be true, and consequently the

local variable will be assigned.

However, the compiler (the little blue robot-figure operating in the blue box) has a different view, where it considers the declaration and uses of the local variable (indicated with dashed lines) to see if it is assigned before it is used. When considering this, the compiler can not determine the value of the condition<sup>1</sup> (the black box with the question mark), and as a consequence it reports an error where the local variable is used. If we let the javac compile process the code example via its command line interface, we get the following result, providing a message and pointing to a position:

```
VariableNotInitialised.java:14: error: variable x might not have been initialized
    return x + y;
           ^
1 error
```

This is where the interaction with javac ends - we get no more assistance and the burden of determining that the fix may be to assign a value declared on line 8

<sup>1</sup>The compiler is performing a static analysis without running the code. One consequence is that it has little knowledge about values of variables and results of expressions. It could possibly infer the result of the condition in the example code but this kind of analysis is typically not done by compilers for the Java language.

is on us. If we consider this as a communication breakdown, how can we provide assistance for repair?

With this in mind, we have developed a prototype for the exploration of the provenance of errors that extracts extra information from the compiler about the source of an error and presents it to the user. Given this relative wealth of information, we will discuss how it can be used to repair a breakdown in understanding and ask the question “how would the interaction look if we explicitly designed for mitigation of breakdowns in the conversational alignment between the developer and the compiler, as well as support for repair?” - In other words, can we design a tool that enables the user to engage in “programming by errors”?

## 2 Background

For ease of access to additional information within the compiler, we decided to build a small extension of a compiler based on a reference attributed grammar (RAG). An explanation of this formalism will be provided in the following section, as well as the motivation behind our selection of a RAG-based compiler as a basis for the Progger system.

### 2.1 Attribute Grammars

The formalism of *attribute grammars* (AGs), as introduced by Knuth [Knu68], is a means of extending a context-free grammar to allow for a declarative description of context-sensitive elements of the grammar. This is achieved by attaching *attributes* to non-terminal nodes of the abstract syntax tree with rules for their evaluation, with attributes categorised as either *synthesised* or *inherited* depending on whether they are used to propagate information upwards or downwards through the tree respectively. For example, consider a simple context-free grammar describing addition and subtraction expressions. The notation used here is a simplified form of the JastAdd notation used to specify the abstract syntax tree (AST) model. Some notation from the concrete grammar is also used for clarity, such as the explicit inclusion of ‘+’ and ‘-’ tokens which would normally be omitted from the abstract grammar. Certain object-oriented concepts are used, such as abstract classes and the use of inheritance, denoted here in the form `<Subclass>: <Superclass>`.

```
abstract Expr
Add: Expr -> Left:Expr + Right:Expr
Sub: Expr -> Left:Expr - Right:Expr
Numeral: Expr -> N
```

Where  $\mathbb{N}$  is the set of all natural numbers,  $\mathbb{N} = 0, 1, 2, 3, 4, \dots$ . To represent the numerical value of an expression, a *synthesised attribute* (val) may be attached to each of the **Expr** nodes and a corresponding *equation* defined for each production in the grammar as follows:



```

syn int Expr.value();2
eq Add.value() = getLeft().value() + getRight().value();
eq Sub.value() = getLeft().value() - getRight().value();
eq Numeral.value() = N;

```

Given the input string “1+2-3” within this grammar, the syntax tree in Figure 2 may be derived with the appropriate attribute values as defined by these equations.

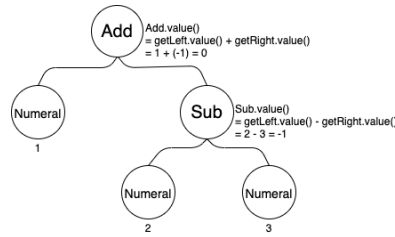


Figure 2: Derived abstract syntax tree.

using *inherited* attributes, where the equation is defined in an ancestor of the node containing the attribute. This will be explained in greater detail in the next section.

The *synthesised* attribute, *value*, can be seen to be calculated based on information provided by the child of the node it is attached to, thereby propagating information up the tree. The evaluation of attributes occurs on-demand, therefore the values of the child nodes would not be calculated until the result is required by the parent node. It is also of interest to propagate information downwards through the tree, however. This is achieved using

## 2.2 Reference Attribute Grammars

The AG formalism has been further extended by the introduction of *reference attributed grammars* (RAGs) as described by Hedin [Hed00]. The primary addition in this extension is to facilitate the referencing of objects by attributes, thereby allowing a *reference attribute* to form a link between one node of the tree and another node at an arbitrary distance from it. This allows for multiple benefits, such as the superposition of graphs over chains of use-def relationships or inheritance structures.

Consider an extension to the previous example that allows for the assignment of values to variables and a predefined print function. This can be achieved by introducing the concept of a block (**Block**) composed of a list of statements (**Stmt**). A statement may be a variable declaration (**Decl**), an assignment of a value to a previously declared variable (**Assign**), a call to print the result of an expression (**Print**), or an expression (**Expr**). We also introduce a Use node to represent calling a variable by reference, and let ID correspond to a string of arbitrary length:

```

abstract Stmt          abstract Expr
Block: Stmt -> Stmt*   Add: Expr -> Left:Expr Right:Expr
Decl: Stmt -> ID Expr  Sub: Expr -> Left:Expr Right:Expr

```

<sup>2</sup>The notation presented here is a simplified form of the JastAdd syntax. A detailed description can be found in the JastAdd reference manual: <https://jastadd.cs.lth.se/web/documentation/reference-manual.php>

```

Assign: Stmt -> ID Expr  Use: Expr -> ID
Print:  Stmt -> Expr    Number: Expr -> N
    
```

In order to associate variable uses with their declarations, we must introduce several new attributes. All **Stmt** nodes will require an attribute (*declares*), that, given an ID as a parameter, will return true if evaluated to a **Decl** statement with a matching ID node, or false for all other **Stmt** nodes:

```

syn boolean Stmt.declares(String id) = false;
eq Decl.declares(String id) = getID().equals(id);
    
```

Another equation may now be defined on **Use** nodes, *decl*, that can be used to calculate a reference to the variable declaration and thereby obtain its value. This may be accomplished by the use of an *inherited attribute*, *lookup*, which we can use to implement the lookup pattern which was previously defined for JastAdd-style RAGs [FSH20]. This attribute is attached to the **Use** node, however its equation is defined further up the tree, within the **Block** node:

```

syn Use.value() = decl.val();
syn Decl Use.decl() = lookup(getID());
inh Decl Use.lookup(String id);
inh Decl Block.lookup(String id);
eq Block.getStmt(int i).lookup(String id) = {
    for (Stmt s : getStmts()) {
        if (s.declares(id)) {
            return s;
        }
    }
    return super.lookup(id);
}^3
    
```

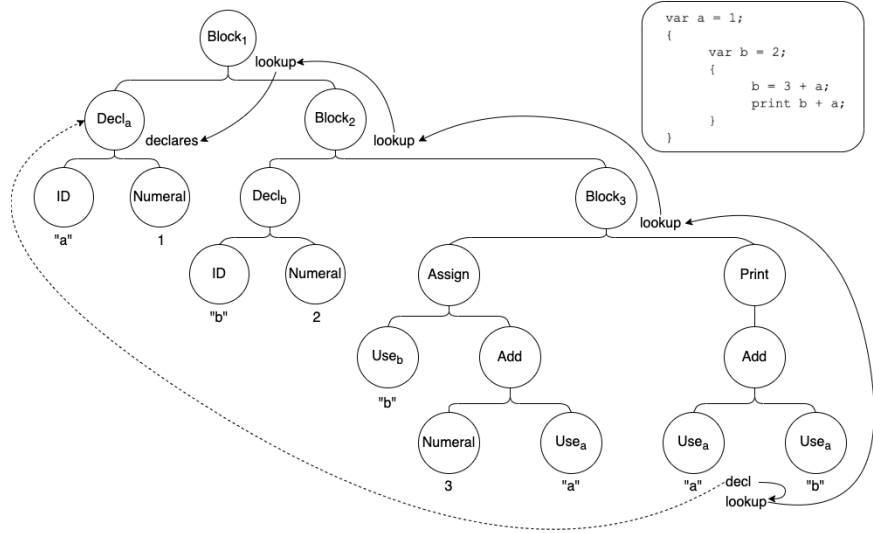
As several attributes are dependent upon each other in this example, the dynamic dependencies must be calculated on-demand when a reference attribute is evaluated. An evaluation stack is used during this calculation which can effectively be compared to a call stack, where an attribute pushed on to the top of the stack can be understood as having been called by the attribute immediately below it.

Consider an input to the compiler in the form of a small program and the resulting syntax tree as shown in Figure 3, with additional information included to highlight dependencies. In the interest of clarity only the dependencies for a single attribute are presented: those which can be used to return the matching declaration for the last reference to “a” in the line: “print b + a;”.

The ability of inherited attributes to propagate information down the tree is highlighted here. The ID value “a” is first propagated up through the **Block** nodes

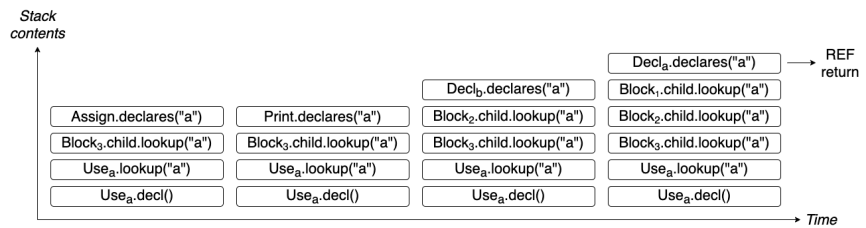
<sup>3</sup>As can be seen, this code snippet includes some imperative code. This is considered to be valid within the JastAdd system, as long as there are no externally visible side-effects.

<sup>4</sup>The subscript numerals 1, 2, and 3 on the **Block** nodes in this diagram correspond to the level of nesting that the respective **Block** occurs at. Subscripts a and b on **Decl** and **Use** nodes refer to the variable name that is declared or used.



**Figure 3:** Extended abstract syntax tree with arrows showing dependencies.<sup>4</sup>

via the `lookup(String id)` attribute until a local declaration can be found. A reference to the **Decl** node is then passed back down the tree and assigned to the `decl()` attribute in the **Use** node. The call stack for this `Use.decl` attribute instance is presented in Figure 4, where the stack is shown at points in the evaluation immediately preceding the top-most object on the stack being popped. As the computation and evaluation of attributes can be expensive in a large syntax tree, caching has also been introduced in order to increase the efficiency of subsequent calls to an already accessed attribute. It is assumed for this example that none of the attributes on the evaluation stack have been previously cached.



**Figure 4:** Evaluation stack.

Reference attributes can be used for various semantic analyses, for instance, in the evaluation of compile time errors. RAGs have also been extended with several

additional concepts, for instance, that of the *collection attribute* [MEH09]. These are attributes where the value is defined by a combination of contributions from other nodes within the tree. Other extensions include circular attributes, rewrites, higher-order attributes, and incremental evaluation, however as they are not pertinent to this paper they will not be discussed here.

## 2.3 The JastAdd Systems

The efficacy of RAGs has been demonstrated by their implementation in the meta-compilation system JastAdd [HM03] and the subsequent implementation of, for instance, the extensible Java compiler, ExtendJ [EH07a], built upon the JastAdd system. Due to the extensibility and modularity of JastAdd, the resultant implementation of the Java specification in ExtendJ provides a convenient entry point to attribute evaluation within a full Java compiler, and is therefore a valuable tool when seeking greater insight into error provenance.

The JastAdd system supports *aspect-oriented programming* [Kic+97], which is reflected in the organisation of the Java specification implementation within ExtendJ. Extension of existing classes within the AST, and the addition of new ones, is supported by the use of inter-type declarations in JastAdd aspect modules, defined in files using the .jrag extension. Aspects use a Java-like syntax to allow for additional classes to be defined within the AST while *attributes* weave additional code into existing generated class files. Aspects are used to group common behaviour together under an easily recognisable descriptor - for example, type checking behaviour within the ExtendJ compiler is grouped together in the Type-Check.jrag aspect file.

The tracing system within JastAdd, first introduced by Söderberg & Hedin [SH10], provides trace events generated at various stages of compilation to perform this attribute tracking. For the previous example in Figure 4, this may be illustrated as in Figure 5 by including the trace events generated at each stage of the stack operations. A `TOKEN_READ` event is first generated upon scanning of the token. Subsequently, as attribute evaluation occurs, a `COMPUTE_BEGIN` is generated as each new attribute is placed on the stack. Once the reference is calculated, the attributes are popped one by one, generating a `COMPUTE_END` event as they are to signify that a value has been obtained. At any point in the evaluation an attribute that has been pre-calculated may have its value read from the cache, at which point a `CACHE_READ` event would be triggered instead.

One benefit of using a RAG-based compiler to build the prototype is the aspect-oriented nature of JastAdd, which groups computations by behaviour, the utility of which we will elaborate on in Section 3.3. Another benefit is the evaluation and subsequent tracing mechanism that is available. By being able to hook into the evaluation stack, this allows us to present to the user not only an error message, as in a conventional system, but also a tree structure showing exactly where in the code the compiler was looking when the error occurred.

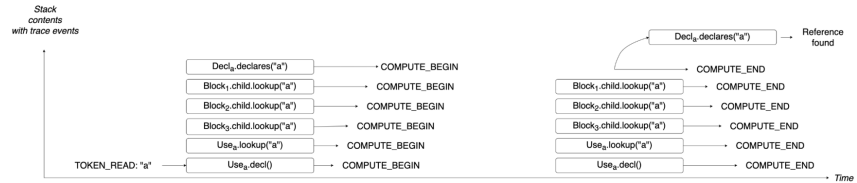


Figure 5: Evaluation stack with trace events.

### 3 The Progger Prototype

In this section, we will explain how we designed the prototype. A literal application of the conversational metaphor in this design would result in an interaction that was similar to a conversational agent, whilst interesting as a possibility this would create a very significant implementation challenge to avoid uncanny valley effects [MMK12]. Instead we aim here to implicitly support the conversational nature of the interaction outlined in the breakdown and repair properties in Section 1. We present the prototype in terms of its client-server architecture (Section 3.1), how we extract error details in the server (Section 3.2), and then how we bring out the details from the compiler to the user in the client (Section 3.3).

#### 3.1 Architecture

To facilitate experimentation in conversational compilers, we elected to build an architecture based on the client-server model, illustrated in Figure 6. The client is a simple web application with a file picker which allows the user to select a .java file, which is then rendered in the client with some simple syntax highlighting to emphasise Java keywords. When the “Compile” button is clicked, the file is uploaded to the compiler service via REST which then compiles the source code and returns a data structure with key information accessed during compilation and the corresponding token locations in the code.

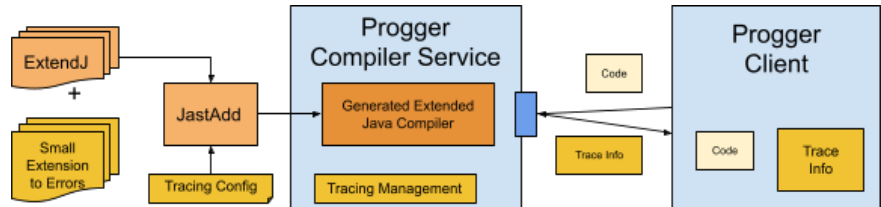


Figure 6: Architecture overview of the Progger prototype.

### 3.2 The Progger Compiler Service

To extract and present a greater amount of information with regards to error provenance, it is necessary to establish an understanding of what information the compiler was making use of when it encountered the error. For this purpose, a Java compiler based on RAGs was selected. Where a non-RAG based compiler may give us access to just the call stack, RAG based compilers make use of declaratively defined objects called *attributes* to perform computation, and consequently can allow access to the attribute evaluation stack. This presents a finer level of information about the computation, with dependencies between units of computation being exposed.

In the RAG-based compiler that we have selected, these computation units are also grouped into aspects, which are clearly labelled collections of attributes and behaviour. This allows us to link operations on the attribute evaluation stack to logically named stages of the compilation process, and thus to map these operations to conversational statements that allow for a greater degree of exploration by users regardless of their level of knowledge about compilers. The following sections will give an overview of RAGs, as well as the attribute grammars that act as their foundation.

#### Extracting Compiler Trace Details

During development of the prototype, the tracing system in JastAdd was updated to contain additional *aspect* information within the trace events, as well as `TOKEN_READ` and several events related to the evaluation of collection contributions. As attributes are defined within aspects, `COMPUTE` events generated by the tracer are now able to report the name of the aspect from which the attribute that is being computed originates. This allows us to link more generic attribute names to the context in which they are being accessed - any attribute defined in the `TypeCheck` aspect, for example, may be easily inferred to have been accessed by the compiler when checking the type of an object.

A key component of the prototype described in this paper is the ability to “hook into” a compiler in order to extract more information relating to errors. Specifically, within ExtendJ a “problems” *collection attribute* is defined in the root node of each compilation unit, which is typically a representation of the code within a Java file. Upon failure of some check in the compiler, a `Problem` object is created and contributed to the problems collection containing information such as an error message, location where the error was discovered, severity etc. By tracking the attributes that are evaluated in the calculation of a contribution to the problems collection, it becomes possible to link an error to various contributing locations in the code.

The prototype achieves this by listening for events triggered at the beginning of a collection contribution check, signified by a `CONTRIBUTION_CHECK_BEGIN` trace event. For example, a node in the tree may contribute a `Problem` to the prob-

lems collection in the event that a return type does not match the method signature. Upon reaching the return statement in a method block, the compiler will begin evaluating this contribution and trigger a trace event signifying so. Once a contribution check begins, we start constructing a tree of the attributes that the contribution evaluation is dependent upon. To do this, any other attribute that is calculated or that has its cached value accessed in the process of evaluation is stored in an internal data structure. At the conclusion of the contribution check a different event is generated depending on whether or not the contribution condition is matched. In the event of a match, the dependency tree is saved and returned to the application once compilation is complete, otherwise it is simply discarded.

### Service REST API

Progger makes use of a simple REST service built upon the lightweight Spark framework<sup>5</sup> to convey the results of a compilation back to the client. This consists of an array of any errors encountered by the compiler mapped to a JSON format as follows:

```
{
  "message": [compiler error message],
  "fileName": [file name],
  "location": [line number],
  "severity": [warning/error],
  "rootNote": [originating attribute node]
}
```

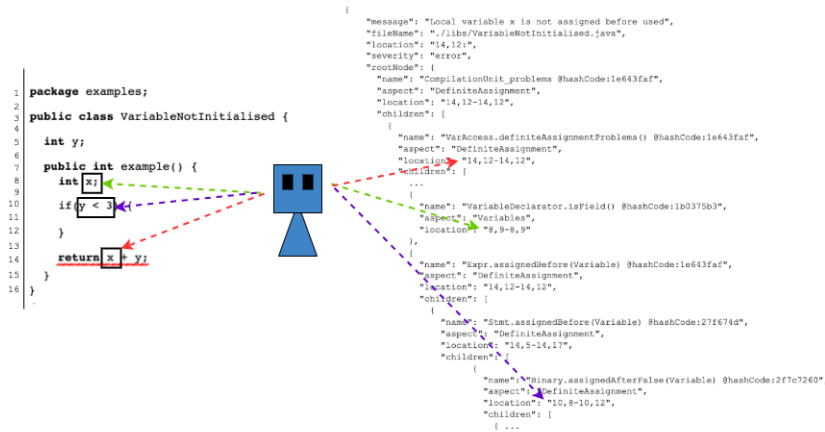
The root serves as the starting point for the attribute dependency tree, with each node containing the following information:

```
{
  "name": [attribute name],
  "aspect": [name of aspect the attribute is associated with],
  "location": [token locations associated with the attribute],
  "children": [array of child attribute notes]
}
```

The location data of attributes within the dependency tree can be used to render annotations over the source code within the development environment. These annotations show us exactly where the compiler was “looking”, in terms of the lexical tokens in relation to the AST nodes hosting the attribute instance, when it encountered the error (illustrated in Figure 7) - effectively a visual representation of the code in a manner similar to that seen by the compiler robot in Figure 1.

---

<sup>5</sup><https://sparkjava.com/>



**Figure 7:** Example JSON excerpts of an attribute error tree. The coloured lines indicate which sections of the source code each attribute refers to.

### 3.3 The Progger Client

The Progger client is implemented as a web UI. Appendix A, Figure 8 shows a screenshot with the example presented in Figure 1. Again we have the error from earlier (yet with a different formulation of the error message due to a different Java compiler being used “under the hood”), but now we also have a “*Tell me more about this*” button. If this button is clicked, we can get more details about what the compiler was considering (approximately the dashed blue arrows in Figure 1) in Appendix A, Figure 9.

Here, we are seeing that the compiler considered several aspects of the code while trying to investigate whether the local variable had been assigned before its first use (`definiteAssignmentProblems`). As we hover over the “error details” to the right, the code related to the aspect being considered is highlighted in the code to the left, for instance, at one point (marked by the cursor) the compiler considered whether the local variable declaration was a field (`isField`).

The error nodes provided at the top level of the JSON output from the compiler service act as a starting point for the visualisation of a conversational interaction with the compiler. The error is rendered to the right of the code in an information box, with the line of code corresponding to the error underlined in red when the error element is moused over. At first glance, the meaning of the error may be unclear to an inexperienced programmer, and it may be useful for them to be able to ask a question of the compiler to help align their mental model - for example



“What were you doing when you encountered this problem?”. To facilitate this discussion, the client provides an option on the error element to *“Tell me more about this”* (Figure 8).

On click, the first layer of the attribute tree is expanded and displayed to the user (Figure 9). In keeping with the conversational tone, the aspect information that is supported in JastAdd-style RAGs are used to more clearly explain what the compiler was doing at each stage of the evaluation process. Since aspects are used to group attributes by common functionality, we have mapped each aspect name to a conversational statement giving a general overview of the purpose of that aspect. In this example, the compiler encountered a problem while checking the definite assignment of  $x + y$  to the return value of the method. This problem was flagged in the `DefiniteAssignment` aspect, which is mapped to a clarifying statement in the client: “I was checking if this definite assignment is valid”, shown in Appendix A, Figure 10. This “error details box” can then be clicked for more details, at which time additional information from the trace is displayed (Figure 9) and can be explored by further expansion of the error details boxes.

With this presentation of the error details, the interaction continues after the error is presented. When the error does not make sense (conversation breaks down) the developer can ask for more information (“tell me more about this”) which then results in a display of a list of the steps the compiler took to detect the error. The user can follow the “train of thought” of the compiler by following along the list of error details and hover over the boxes to see what parts of the code that were considered, while also considering the names of the aspect and attribute of the computational unit.

Appendix A, 11 includes the end of the error details from Figure 8 along with the code highlighting connected to those error detail boxes. As the user gradually hovers over the list of error details (all concerned with “definite assignment”) from top to bottom, the “return statement” is highlighted (where the error is marked), then the if statement, the assignment, and then the condition. The final box then highlights the declaration. The compiler is trying to determine whether the “x” variable has been assigned. With the highlighting we can follow along to the areas in the code (away from the error location) that it had to consider.

## 4 Discussion and Implications for Future Work

This paper outlines a strategy for making the interaction around errors a foreground part of the experience of a developer. In doing so it practically demonstrates the work that is needed in order to create an environment with a closer alignment between a developer and a compiler. This work proceeds on two fronts. Firstly, the internal processes and data structures of the compiler have to be exposed, and, where possible, mappings created on top of them that are likely to be closer to the model by which the developer thinks of what is happening in the compiler. Sec-

ondly, the user interface by which the developer interacts has to be brought closer to the compiler, with additional elements added to enable requests for additional information in particular cases. This engineering work to ‘meet in the middle’ needs to be built on top of an architecture that can support allocation of different pieces of work to the various components in the system. Doing this results in a more conversation-like interaction with the compiler shifting away from an idempotent input/output model, to a question/answer model, and in doing so highlights a number of possibilities for future work.

In order to empirically characterise the effects of this to a more conversational interaction mode it will be necessary to support a wider range of features within Progger, for example saving changes, introducing syntax highlighting and multiple files in order to create a more representative interaction context. More significantly, there are further refinements that can be made to the presentation of the errors themselves, hiding extraneous information and supporting the developer using Progger by focussing on the conversational interaction. Once these improvements have been completed we propose to study in a representative commercial context how developers go about doing their work when the focus is shifted to a conversational interaction around error messages.

The tool in its current state may also lend itself well to an educational context. While the information presented in the attribute tree view may not necessarily solve a problem by itself, it will point the user towards relevant places in the code - potentially highlighting to novice programmers the most relevant areas for review when trying to resolve an error. In this way, Progger acts not as a problem-solving tool, in that it does not actively suggest fixes, but rather as one that aids our understanding and facilitates learning.

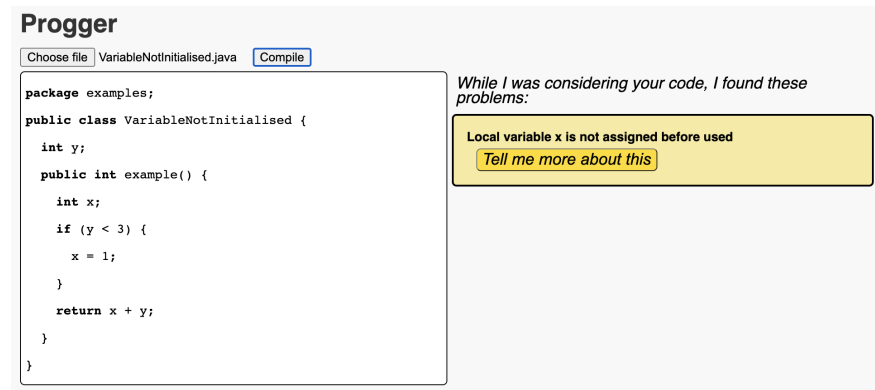
## 5 Acknowledgements

This work is supported by the Swedish Foundation for Strategic Research under Grant No. FFL18-0231 and the Swedish Research Council under Grant No. 2019-05658.

## Appendix A: Screenshots

## References

- [Bec+19] B. A. Becker et al. “Compiler error messages considered unhelpful: The landscape of text-based programming error message research”. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR)*. 2019, pp. 177–210.



**Figure 8:** Screenshot of the Progger prototype with code to the left and error information to the right.

- [Ben+19] E. Beneteau et al. “Communication breakdowns between families and Alexa”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI ’19. 2019, pp. 1–13.
- [CSM21] L. Church, E. Söderberg, and A. T. McCabe. “Breaking down and making up-a lens for conversing with compilers”. In: *Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2021.
- [DP09] H. Dubberly and P. Pangaro. “What is conversation? How can we design for effective conversation”. In: *Interactions Magazine* 16.4 (2009), pp. 22–28.
- [EH07a] T. Ekman and G. Hedin. “The JastAdd system—modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.
- [FSH20] N. Fors, E. Söderberg, and G. Hedin. “Principles and patterns of jastadd-style reference attribute grammars”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. 2020, pp. 86–100.
- [Hed00] G. Hedin. “Reference attributed grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.
- [HM03] G. Hedin and E. Magnusson. “JastAdd - An aspect-oriented compiler construction system”. In: *Science of Computer Programming* 47 (2003), pp. 37–58.
- [HH11] A. Henderson and J. Harris. “Conversational Alignment”. In: *Interactions* 18.3 (2011), 75–79.

- [Kic+97] G. Kiczales et al. “Aspect-oriented programming”. In: *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP)*. Springer. 1997, pp. 220–242.
- [Knu68] D. Knuth. “Semantics of context-free languages”. In: *Mathematical Systems Theory* 2.2 (1968), pp. 127–145.
- [MEH09] E. Magnusson, T. Ekman, and G. Hedin. “Demand-driven evaluation of collection attributes”. In: *Automated Software Engineering* 16.2 (2009), pp. 291–322.
- [MMK12] M. Mori, K. F. MacDorman, and N. Kageki. “The Uncanny Valley [From the Field]”. In: *IEEE Robotics Automation Magazine* 19.2 (2012), pp. 98–100.
- [SH10] E. Söderberg and G. Hedin. “Automated selective caching for reference attribute grammars”. In: *Proceedings of the 3rd International Conference on Software Language Engineering (SLE)*. Springer. 2010, pp. 2–21.

**Progger**

Choose file VariableNotInitialised.java Compile

```

package examples;
public class VariableNotInitialised {
    int y;
    public int example() {
        int x;
        if (y < 3) {
            x = 1;
        }
        return x + y;
    }
}

```

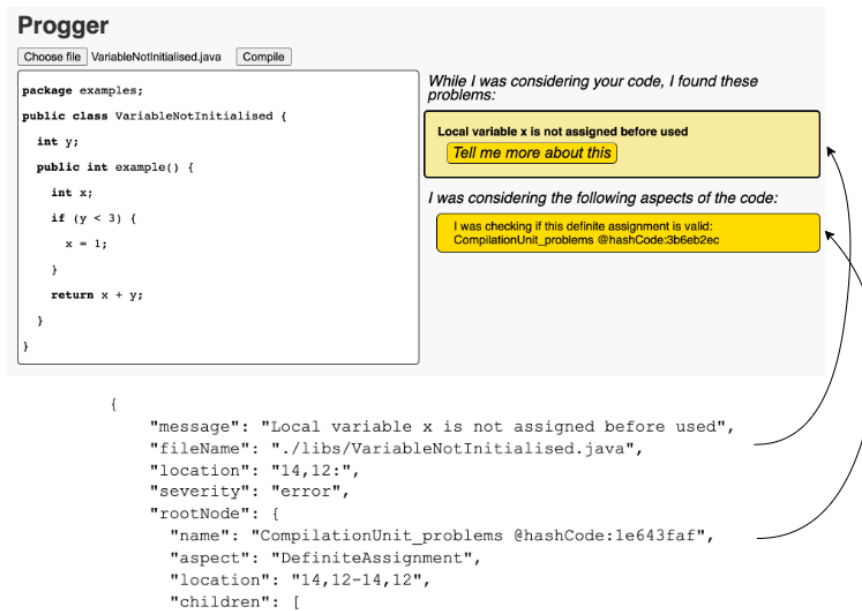
While I was considering your code, I found these problems:

**Local variable x is not assigned before used**  
[Tell me more about this](#)

I was considering the following aspects of the code:

- I was checking if this definite assignment is valid: CompilationUnit\_problems @hashCode:3b6eb2ec
- I was checking if this definite assignment is valid: VarAccess.definiteAssignmentProblems() @hashCode:3b6eb2ec
- I was checking if this definite assignment is valid: Expr.isSource() @hashCode:3b6eb2ec
- I was propagating a variable's scope: VarAccess.decl() @hashCode:3b6eb2ec
- I was propagating a variable's scope: VarAccess.decl() @hashCode:3b6eb2ec
- I was propagating a variable's scope: VarAccess.decls() @hashCode:3b6eb2ec
- I checked the attributes of a variable: VariableDeclarator.isField() @hashCode:66d33a
- I was checking if this definite assignment is valid: Declarator.isValue() @hashCode:66d33a
- I was checking if this definite assignment is valid: Declarator.isBlankFinal() @hashCode:66d33a
- I was checking if this definite assignment is valid: Expr.assignedBefore(Variable) @hashCode:3b6eb2ec
- I considered a rewritten version of the model: ParseName.rewrittenNode() @hashCode:1e643faf
- I was checking if this definite assignment is valid: Stmt.assignedBefore(Variable) @hashCode:7dc36524
- I checked the attributes of a variable: VariableDeclarator.name() @hashCode:66d33a
- I was building the class path: CompilationUnit.relativeName() @hashCode:16e8568
- I was checking if this definite assignment is valid: Expr.isDest() @hashCode:3b6eb2ec

**Figure 9:** Screenshot of the Progger prototype with the error details from Figure 8 expanded. The variable, x, highlighted in the code pane corresponds to the location investigated while evaluating the attribute that the cursor is over.



**Figure 10:** Mapping of errors and attribute details to conversational statements.



**Figure 11:** The error trace details shown in order of occurrence in the computation with arrows pointing to how the highlighting in the code changes on hover over error details.

# VISUAL CUES IN COMPILER CONVERSATIONS

---

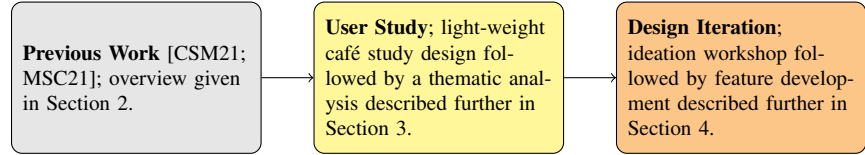
## Abstract

When people are conversing, a key non-verbal aspect of communication is the direction in which the participants are looking, as this may convey where each person's attention is focused. In a programming context, for instance an integrated development environment (IDE), the interaction design frequently directs the programmer's gaze towards specific locations on-screen. For example, syntax highlighting and error messaging may be used to draw attention towards problematic sections of code. However, error messages frequently direct the user towards the compiler's point of discovery as opposed to the actual source of an error. Previously we have applied a conversational lens considering the interaction between the programmer and the compiler as a conversation, in this work we refine that into an "attentional lens". We consider via a prototype and small exploratory user study the difference between where a developer chooses to spend their attention, where the tooling directs it, and how the two might be aligned through the use of visualisation techniques.

## 1 Introduction

For programmers, the act of programming is a primarily one-way relationship: the programmer writes code, most commonly in an IDE; executes it through a compiler; and receives limited feedback in the form of error messages. These error messages are often obtuse and difficult to read [Ben+19], and may even mislead the reader into looking at the wrong sections of code altogether [Bec+19; Kat+09]. This "feedback" is not only limited in form, but is also delivered in a purely binary format - an error occurs, or it does not.





**Figure 1: Overview** of the work presented in this paper and its disposition.

In our previous work, “Breaking down and making up - a lens for conversing with compilers” [CSM21], we explored the consequences of expanding on this interaction to allow for a more complete two-sided relationship between developer and environment. This was achieved by analysing the interaction in the context of a conversation between two participants, inspired by the work of Dubberly & Pangaro [DP09] and Pask [Pas76], thereby applying a “conversational lens” to the activity of programming. For instance, under the conversational lens, the participants of the conversation can be said to be the programmer and their development environment, including IDE, virtual machine, compiler, and various other software components. As stated by Dubberly & Pangaro [DP09], a key aspect of conversation is the mutual construction of meaning and convergence upon agreement - when applied to a programming activity, the meaning of the conversation can be said to be agreed upon when the programmer expects their code to execute in a certain manner, and the compiler performs this execution as desired. This activity helped to highlight areas where programming as an interaction diverged significantly from a familiar human interaction, and instances where the development environment lacks the ability to properly engage in the conversational activity as an equal partner. Based on these findings, a prototype development tool, Progger, was created [MSC21], which will be described in more detail in Section 2.

In this paper, we present the second iteration of our exploration of applying a conversational lens to interactions with compiler error messages. In this iteration we narrow our conversational lens to that of *attention*. In a situation where the conversation between programmer and compiler breaks down, the standard response comes in a textual form. By contrast, in normal human interactions other factors come in to play, such as facial expression, gaze, voice, and body language. These can be used to introduce additional information into the interaction, such as a participant’s disposition, or their focus of attention.

With this in mind, we present the results of our exploration of attention in the form of visual cues in the interaction with the compiler. Figure 1 gives an overview of the different components of the iteration presented in this paper. We present results of a user study evaluating the Progger prototype (Section 3), the results of a follow-up ideation workshop, building on insights from the user study (Section 4), and finally we end the paper with a discussion of design implications of this work (Section 5).

## 2 Background and Related Work

In this section, we cover related work connected to the use of attention, either as focus in an empirical study or as part of an intervention (Section 2.1). We also provide a brief summary of our past work on the Progger tool where we applied a conversational lens to the interaction with error messages (Section 2.2).

### 2.1 Attention in Software Development Tools

Software development is a complex task that requires high cognitive load (CL) [Gon+19]. It also involves a wide variety of tooling, which further adds to that. One CL-intensive activity during this process is reading and understanding code. This often happens in a development environment, with some assistance from the underlying tool(s). The assistance can manifest in multiple representations, for example, textual (e.g., error messages) and visual (e.g., coloring and alignment). In order to digest such multi-modal information, attention is needed from developers.

Based on the assumption that attention resting on code reflects the visual effort developers take to read and understand it, several studies have focused on code comprehension; e.g., [Bus+15; Sto05]; contrasting the behaviours of experienced and novice programmers. Below we selectively elaborate on some work we deem more relevant to this study.

Crosby et al. [CSW02] employed eye tracking to examine the roles that *beacons* play among programmers. A beacon could, for instance, be in the form of a comment beacon or a line of code that contains a hint about program functionality. Experienced programmers were more aware of and inclined to make use of beacons in code to facilitate their reading. Novice programmers were less capable of distinguishing between beacons and other areas of code, and thus made little use of them.

Bednarik [Bed12] analysed the temporal development of visual attention strategies between novices and experts during debugging in a multi-representational development environment. The study found that experts and novices exhibited similar gaze behaviours in the beginning but diverged in the later phases. Experts were more resourceful with the available information while novices monotonously stuck to one strategy. For challenging bugs, experts more actively related the output to code.

The presented work by Crosby et al. and Bednarik indicate that: 1) efficient utilisation of visual cues elevates code comprehension and debugging, and 2) expert and novice programmers need to be treated differently when designing tools for them; in particular, novices may be those who need help most.

Attention-based interventions have been explored in a couple of studies. For instance, Ahrens et al. [ASB19] visualised developers' attention in the form of heat maps and coloured class names in Eclipse. In the context of software maintenance

tasks, they reported that these two mechanisms provided little aid in orientation and code finding for developers, although the heat map slightly alleviated the cognitive demand. Most developers, especially experienced ones, did not find them helpful. Instead, they found the visualisations to be a distraction from understanding the code quickly and clearly.

Another example is the work by Cheng et al. [Che+22]. They empirically evaluated the usefulness of a tool that captures developers' shared gaze in real time. They found that shared visualisation mechanisms such as gaze cursor, area of interest (AOI) border, grey shading, and connected lines between AOIs helped improve the efficiency of code review. Assisted by these visual features, especially the cursor and border, developers found it easier to identify where their collaborator looked and focused. Based on that, they could adopt either a follow- or separated-strategy to find the bugs more quickly.

These two studies by Ahrens et al. and Cheng et al. demonstrate various ways of approaching attention visualisation in a software development context and the possible granularity of how attention can be visualised. Further, we gather that there appears to be no existing best practises for the time being and the design remains a largely open space.

## 2.2 The Evolution of Progger

In conversational theory, participants work collaboratively to create a meaningful shared mental model of the on-going conversation [Pas76]. Frequently, however, the understanding of the participants becomes divergent for some reason. For example, something may be misheard or misunderstood by one actor, or an incorrect assumption may be made about implicit knowledge [Ben+19]. When this occurs, it is said that there is a "breakdown" in the conversation, at which point a meta-conversation must be entered into in order to repair the fault [DP09].

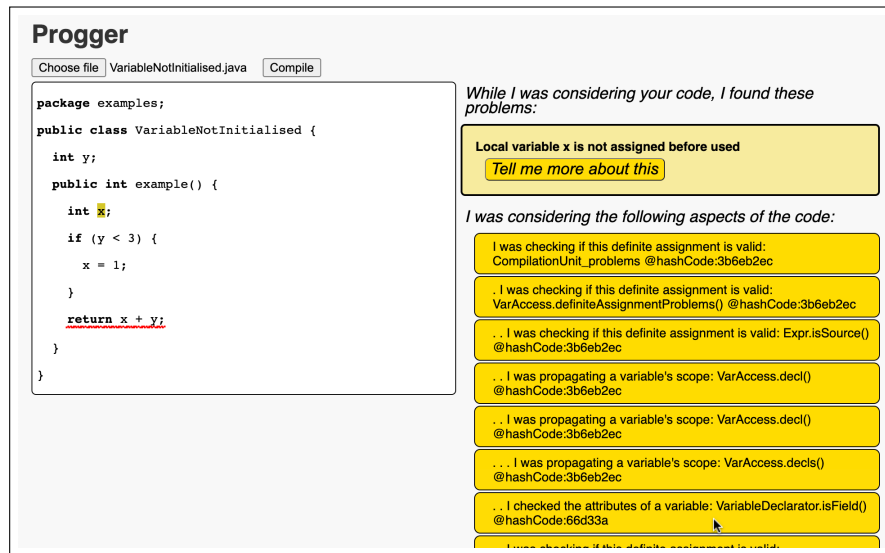
When applying the conversational lens to the activity of programming, the participants become the programmer and their development environment, which may contain several tools such as a compiler, IDE etc. In this context, it can be said that when a program does not perform as expected by the author then a breakdown occurs [CSM21]. When such a breakdown occurs, a common form of feedback to the developer is that of compiler error messages. However, unlike in a natural conversation, the compiler error message is the end of the interaction - if it is not understood by the programmer, there is no mechanism for further exploring the breakdown. At this point the onus of repairing the conversation falls entirely on the human participant.

In an attempt to bridge this gap between programmer and compiler, a research tool was created in the form of a simple web-based Java IDE [MSC21]. This tool, Progger, consists of a Dart<sup>1</sup> front-end communicating via a REST API with a Java compilation server. The compilation service contains a small extension

---

<sup>1</sup>The Dart programming language, <https://dart.dev/>.

to the extendable Java compiler ExtendJ<sup>2</sup> [EH07b], which itself is based on the JastAdd<sup>3</sup> meta-compilation system [EH07a]. The decision to base the tool on ExtendJ was made for a number of reasons, the most significant being the fact that it is a compiler that makes use of reference attribute grammars [Hed00]. An explanation of this formalism is beyond the scope of this paper, however a detailed description of reference attribute grammars and their significance within the Progger system may be found in our previous work, "Progger: Programming by Errors (Work In Progress)" [MSC21].



**Figure 2:** Screenshot of Progger version 1.0. The left side shows a small Java code snippet with an error in it pointed out with a red squiggly line. The right side shows the error with a "Tell me more about this" button which has been clicked here to expand an attribute trace tree shown in the bottom right.

By use the tracing system inherent in JastAdd [SH10], the evaluation of attributes is tracked by Progger. When a compiler error occurs, this evaluation tree is logged and returned to the front-end for display to the user. This tree is then displayed in the development environment as shown in Figure 2, with the error message augmented by a button with which the user can ask the compiler to "tell me more". As many nodes in the attribute tree are directly related to tokens in the text of the code, this information is used to highlight the relevant code sections as the user mouses over the tree. In this way, the user is able to follow the "thought

<sup>2</sup>The Extensible Java Compiler ExtendJ, <https://extendj.org>.

<sup>3</sup>The meta-compilation system JastAdd, <https://jastadd.org>.

process" of the compiler as it looked at various parts of the code in an attempt to validate the line at which the error occurred.

### 3 User Study

In initial testing of the Progger research tool, the development team found the results to be of interest. Despite this interest, it was understood that the tool itself worked on a very simple assumption: providing more information to the user is inherently useful. Much of this information, however, came in a form which was only intelligible to someone with a reasonable understanding of the internal workings of RAG-based compilers. In contrast to this, we felt that the target audience who stood to benefit most from a tool like Progger was that of relatively inexperienced programmers. Due to this disconnect, it was decided to undertake a user study in order to determine the usefulness of the tool in its current state when presented to the target audience.

#### 3.1 Study Design

A light-weight exploratory study, here named a “café study”, was undertaken in an effort to obtain initial design feedback for a relatively low time investment. As a general target audience of fairly novice programmers was selected, the café study was conducted on the grounds of Lund University. This took the form of a booth set up in the foyer of the computer science building, as shown in Figure 3, where students were offered the opportunity to complete a short task within Progger in exchange for a lunch coupon.

Students deciding to participate in the study were asked to complete an informed consent form, after which they were presented with a single-class Java program, chosen randomly from a set of 3 pre-defined programs, each of which contained several compiler errors. These programs were selected randomly from a public repository<sup>4</sup> of solutions to Kattis<sup>5</sup> programs, with a selection of errors manually inserted into the code by the first author. The errors are representative of a small set of semantic error patterns, selected to provide instances where the prototype was found to be particularly helpful in initial testing, and instances where it was not. For example, uninitialised variable error rendered a set of localities that were deemed to be interesting, while missing import statements did not return any locations within the class file. Figure 8 includes an example of a Java code snippet used in the study. Participants were then asked to attempt to fix the errors, to the best of their abilities, while the screen and verbal interactions were recorded. This method yielded a set of 13 recordings over a two day period. Of the set of

<sup>4</sup>Provided with permission by Pedro Contipelli: <https://github.com/PedroContipelli/Kattis>, visited at commit 30884ba

<sup>5</sup>Kattis problem archive: <https://open.kattis.com/>



**Figure 3:** The user study booth.

participants, none of them had industrial programming experience, with most of their exposure to programming coming in an academic context of between 0 and 4 years of higher education. This information was obtained via a follow-up survey distributed by e-mail, which yielded a relatively low response rate. In retrospect, an immediate follow-up survey, to be completed on-site at completion of the study task, would have been a more rigorous method of acquiring this data, a factor that will be taken into consideration in future iterations of the café study methodology.

Ultimately, the study setup was considered to be a success by the authors. Despite the low level of commitment required to set up and conduct the experiment - in the region of hours of total work - the aforementioned assumption, that more information is inherently useful, was effectively challenged by the study findings. These findings will be discussed in detail in the following section.

### 3.2 Data Analysis

After an initial transcribing exercise, the transcripts were read through independently by the first and second author, with the aim of completing a thematic analysis. This entailed the identification of comments and interactions that were deemed to be of interest. Depending on the content of the highlighted excerpts, a number of codes were coalesced during this process. Once the initial reading and codifying of the transcripts was completed, the first and second author met to compare

the annotated transcripts, whereupon areas of overlap were identified and used to inform a combined set of codes, shown as boxes in Figure 4.

Codes represented specific features like highlighting, capturing both positive (e.g., *"It's very good with the highlight system because you know exactly where you want to look initially"*) and negative aspects (e.g., *"It's highlighting everything, it's too much"*), as well as whether the error messages were deemed to be helpful (e.g., *"It's putting human sentences instead of just like error codes and [...] more explained I would say, absolutely more explained"*) or unhelpful (e.g., *"It didn't describe the error very well"*). Another example of a feature represented by a code was the attribute trace tree and, for instance, when it was found to not be helpful (e.g., *"All this text, it doesn't really say anything to me"*). Other codes captured broader aspects such as how beginner friendly the tool was (e.g., *"if you're a beginner programmer and you would get this kind of [...] feedback on your code, it would be very much easier to [...] fix it"*), or comments made when encountering prototype bugs (*"This isn't showing anything"*).

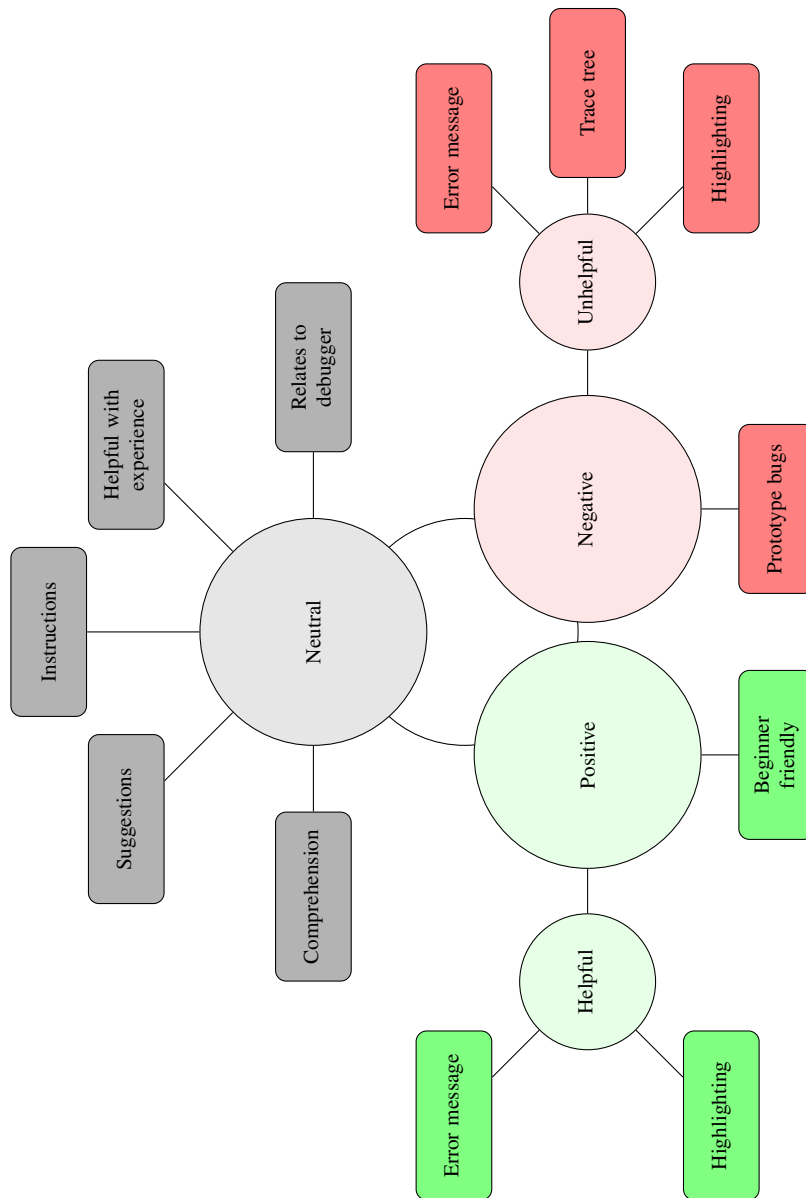
Furthermore, a number of codes were introduced for the purpose of categorising interactions that were deemed to be less interesting, such as: instances where the interview subject is verbalising their process of comprehension (e.g., *"So, I have a couple of 'if' statements, and if none of these are fulfilled I will returned T"*); asking the interviewer for instruction (e.g., *"Do you want me to recompile it?"*); relating Progger to their experience of conventional debuggers (e.g., *"Some debuggers are [...] scary because they have an overwhelming amount of functions that you're not really accustomed to [...] while this one [...] generalises it more by giving you the simple fact of 'this is where we think the problem is'"*); making suggestions for future improvements (e.g., *"One step further could be [...] to make a suggestion how to fix it"*); and relating that the tool may become more useful with experience (e.g., *"If you [...] get to know this tool I think it becomes easier"*).

These codes were then applied to two of the interview transcripts in a collaborative exercise involving the first and second authors, in order to come to an agreement upon interpretation. After completing this exercise, the first author applied the combined code set to the rest of the transcripts individually. Certain codes occurred frequently across all participants, with a breakdown presented in Table 1.

Across the codes, various loose themes began to emerge: neutral discussion, positive comments about the tool, and negative comments about the tool. Within the positive and negative themes, two sub-themes, helpful and unhelpful respectively, were also constructed. The final theme map is presented in Figure 4 with themes shown as circles connected to codes in boxes.

### 3.3 Discussion

Through reading of the transcripts and the thematic analysis exercise, two main take-aways arose: 1) that the trace tree showing the internal workings of the compiler as it traversed the attribute tree was largely deemed to be unhelpful, with



**Figure 4:** Identified **themes**, represented by circles, and associated **codes**, represented by boxes.



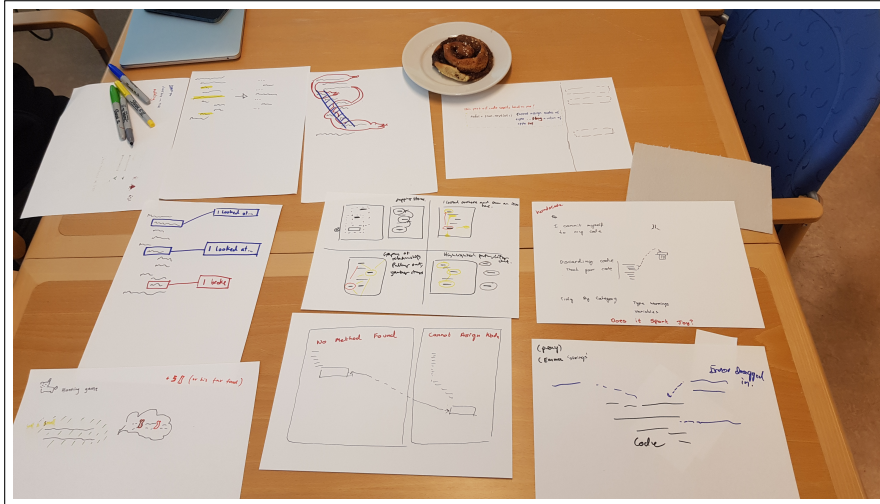
**Table 1:** Overview of occurrence of codes per participant. Colour coding represents the theme which each code ultimately fell into (green for positive, red for negative, and grey for neutral), and intensity in terms of percentage of total words dedicated to each code by a participant (with color intensity increasing with percentage).

Code	Participants												
	1	2	3	4	5	6	7	8	9	10	11	12	13
Beginner friendly	5.5	23.9	25.9				6.4			3.1		7.6	20.1
Helpful Highlighting	20.4			30.3	5.7	17.4	14.6	24.2		14.1	8.7	10.6	11.9
Helpful Message		12.5	4.1	20.8			12.0						7.9
Unhelpful Trace Tree			16.3		4.0		9.1			6.8	19.8		
Unhelpful Highlighting				4.2			3.7	23.2			14.7		
Unhelpful Message	7.5							2.8	4.6				
Prototype Bugs	2.6					10.1				4.0			
Comprehension	42.8		19.7	30.7	44.8	49.3	6.0	22.0	49.1	15.3	8.1	59.1	43.1
Helpful Experience	7.5				12.9				24.1	9.8			
Relation to Debugger									3.3	21.5	17.6	7.6	
Suggestions							26.1	13.1				9.0	
Instructions				5.8			5.3		1.9	5.5			

no positive comments made regarding it, and 2) that the highlighting of localities within the code was of particular interest to participants - when it worked well it was praised highly, when it did not work well it was criticised as distracting. The prevalence of these sentiments is lent credence by the incidence of the relevant codes across the transcripts: out of 13 interviews, 11 participants made mention of the highlighting feature in a positive light, with 5 containing a negative comment. Similarly, the trace tree was mentioned in an unfavourable light in 6 out of the 13 transcripts - the highest incidence of any one negative code - while it was not spoken of positively by any participant.

In light of the related work on attention, the positive feedback regarding *highlighting* drew comparisons to the results of the study by Crosby et al. [CSW02]. In this study, it was found that experts were more aware, and made more use, of beacons in code, while novices were found to not be as adept at distinguishing beacons from other less-relevant code. We hypothesise that the use of highlighting may help to even out this distance by drawing the attention of novices to areas of the code which may act as beacons for experts.

Regarding the dominantly negative feedback about the attribute trace tree, we did not see anything close to an effect where a participant expressed that they understood the compilers "train of thought" by using the trace tree. How we implemented the feature together with the limited user study may be two reasons for this. We further speculate that the effect we saw here may be related to Norman's



**Figure 5:** A selection of design concepts as produced by the ideation workshop.

gulf of evaluation [Nor13].

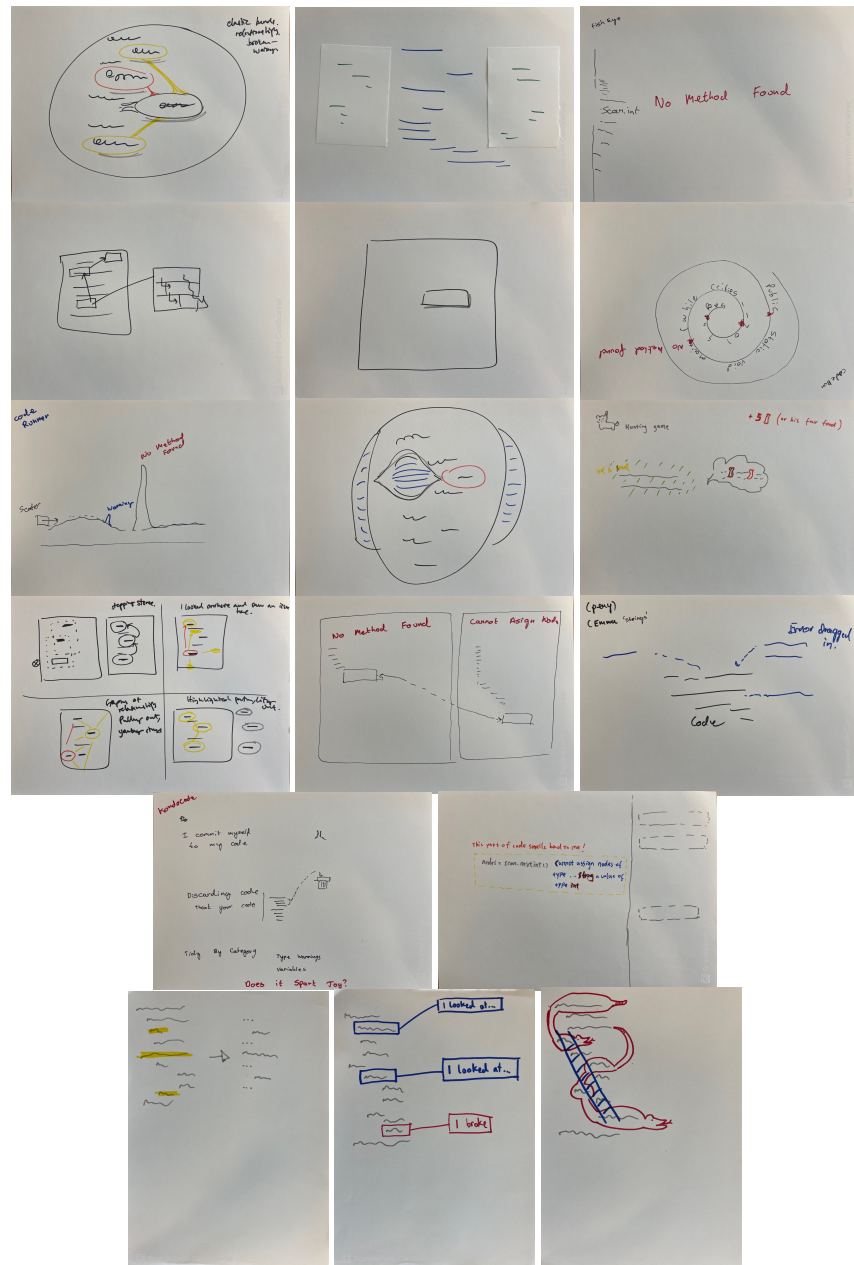
The consequences of these findings were discussed and a design ideation workshop scheduled in order to iterate upon the design of the prototype. The starting point for the design iteration was ultimately decided to be a new version where the trace tree was entirely removed, and where *locality* became the primary means of interaction with the user.

## 4 Design Iteration

In order to operationalise the above data into a design process, we elected to perform a divergent design phase [Cro05; Dub04; Pug81]. During this phase we attempted to generate as many broadly differing designs as possible in order to create ideas that we could curate. These were done in the context of the original design, but with the explicit intention of not being literally driven by it.

The design workshop was run in two phases with all the authors creating one series of new designs, then a short presentation where each of the authors described their ideas to each other, followed by another iteration to allow cross-pollination of ideas, followed by a final discussion and wrap up.

The process was successful in generating a wide range of different designs and interaction metaphors, which can be seen in Figure 6.



**Figure 6:** A collage of the ideas conceptualised during the ideation session.

## 4.1 Data Analysis

After the main ideation workshop session, a period of time was allowed for the participants to consider the proposed ideas. Ultimately, the first author selected three concepts for further development, based on both novelty and technical feasibility. From these three design concepts, storyboards were drawn up to further illustrate the design interaction. These storyboards are presented in Figure 7.

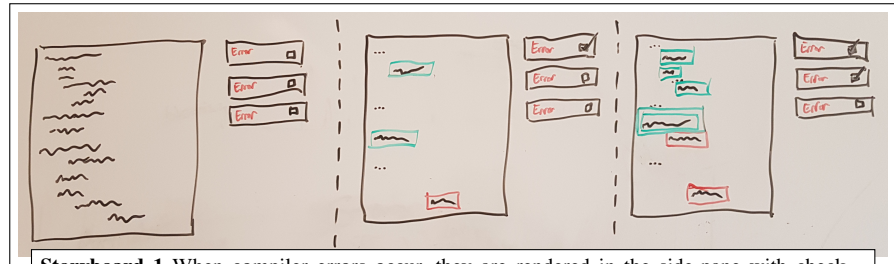
Following the creation of these storyboards, a final session was held between the first three authors to narrow the selection down to a single final concept for further development. Locality, one of the main takeaways from the user study, was found to be a strong theme in each of the concepts, taking the form of obfuscating non-relevant code in Storyboards 1 and 3 and the physical separation of relevant code from the main body in Storyboard 2. Storyboard 2, however, also introduced a physical relationship between the elements of the code. As described in Figure 7, it was conceptualised that the information collected during the evaluation of an error, specifically the token locations in the code, could be used to introduce a "weight" to each considered element. This "weight" would be based on the number of times the compiler passed over each token - in essence, a compiler "heatmap" - and led to a discussion amongst the authors about what this information might reveal. Ultimately, three theories emerged:

1. Locations that are *frequently* revisited by the compiler may indicate sections of code that are critical to the calculation of an error.
2. Locations that are *less-frequently* visited by the compiler may indicate sections of code that the compiler struggles to parse correctly, leading to it being visited very few times before an error is thrown.
3. Frequency of visitation of the compiler may have no significance when considering the source of an error.

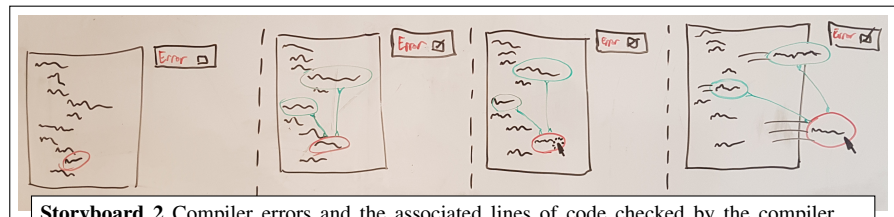
As no consensus could be arrived at which of these three theories was most likely to prove correct, it was decided that Storyboard 2 offered the most interesting opportunity for exploration. For this reason, Storyboard 2 was selected for further development.

## 4.2 Implementation

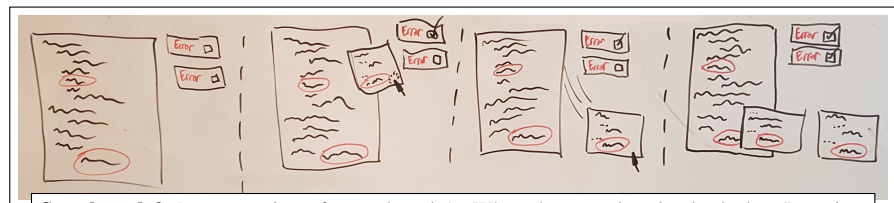
Building on the previous Progger work, a new version of the prototype was developed. The initial version of Progger, as described in Section 2, uses the JastAdd tracing system to track the evaluation of attributes at the time of an error occurring. From this, an evaluation tree is constructed, with many of the nodes directly related to token locations in the code. As token locality was the primary focus of the most recent iteration, this previous design meant that no further information was required to be extracted from the compiler in order to determine the heatmap.



**Storyboard 1** When compiler errors occur, they are rendered in the side-pane with check-boxes. When the box for a particular error message is checked, all code not directly checked by the compiler is obfuscated. Multiple errors may be checked, with lines of common interest highlighted.

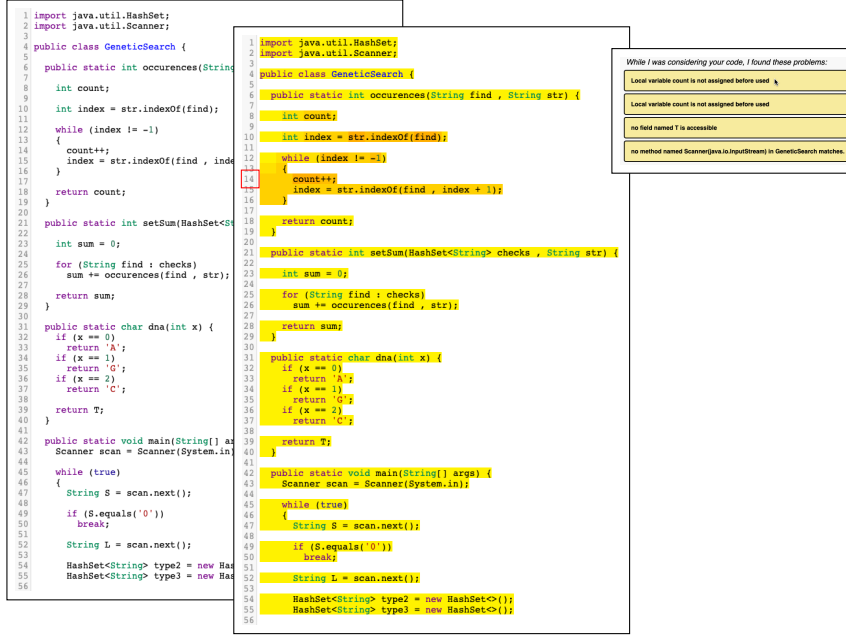


**Storyboard 2** Compiler errors and the associated lines of code checked by the compiler are highlighted with ellipses, with "elastic bands" connecting them. When the error node is clicked and dragged, the associated code sections are pulled out of the code pane alongside it. The number of times a code fragment is checked by the compiler is used to assign "weight" to the elements, with more frequently checked elements "sticking" to the code pane.



**Storyboard 3** A permutation of story board 1. When the error box is checked, a "post-it note" element is added to the display, showing only the code that is directly checked by the compiler. These post-it notes may be moved around the screen at will.

**Figure 7:** Storyboards of expanded concepts from the ideation session.



**Figure 8:** An example of a heat map generated on a one of the Java code snippet that was presented in the user study. As the mouse pointer is hovering over the first error ("local variable count is not assigned before use" on line 14) the attention of the compiler when finding that error is shown is visualized as a line-based heat map.

On reception of the attribute tree from the compiler, Progger v2.0 iterates through the tree and logs each instance of a "location" attribute occurrence. These location attributes come as a range, e.g: 14, 0–15, 12, which is of the format *startLine, startColumn–endLine, endColumn*. From this list of ranges, a map is calculated where the key is a *single location*, e.g. 14, 0, and the value is the number of times this token is visited across all location ranges. From this map, highlighting can be applied to the code pane, with the darkness of the highlighted code calculated from the number of times the token has been visited by the compiler. An example of this is found in Figure 8.

## 5 Discussion

In this paper, we have presented the results of a user study evaluating the approach implemented in the Progger prototype [MSC21]. We used a light-weight café style

study, combined with a thematic analysis of transcripts, to gather design input for an ideation workshop. The results from the user study played down the utility of the attribute trace tree (which we had some hope for) while the utility of the companion highlighting was brought to the surface. As a consequence, our focus was geared toward that of attention and the role it plays in the kind of "compiler conversations" we are considering in this work. With input from the sketches generated in the ideation workshop, we explored one design direction focusing on incorporating visual cues into Progger in the form of a "compiler attention heat map" laid out on visual tokens in the source code.

**Heatmaps & Attention** As mentioned in Section 2, Ahrens et al. [ASB19] have also explored the use of heat maps but with the goal of visualizing the attention of other developers. They found some issues in their method due to imprecision between the generated heat maps and the mapping to the source code, distorting the locality of the attention, causing some of the experienced programmers in their study to find the visualisation technique distracting. In our explored setup, we consider the "thought-process" of the compiler where the heat map weights are calculated and assigned based on the number of code element visits by the compiler during analysis. We speculate that we would not see the same distortion of attention as when eye-tracking data is mapped to code lines as in the case with the work by Ahrens et al., but we may on the other hand see distortions amounting from the structure of the abstract syntax tree modelling the code.

**Collaboration & Attention** We find the work by Cheng et al. [Che+22], which explores visualisation of other developers' gaze in a collaborative setting, inspiring. It may be worthwhile to explore a combination of the conversational lens, as we are applying here, in a similar setting. For instance, questions like *"how can conversations within one group help another group?"* or *"how can the compiler's knowledge about experts enhance its communication with novices?"* could be considered. As a possible exploration, we can imagine the compiler as a host that is able to store all programming mistakes made, and visual attention given by, developers. When a new actor enters the environment, the most frequently looked parts of code or the most possible problematic code regions are already marked out. In that sense, there is a historical component to the conversation where past actors remain present in the new conversation.

**Programmers' Attention** Earlier work on Attention Investment [Bla02] within the PPIG community has explored the way in which programmers considered the likely costs and rewards of expenditure of their attention with a notational system. In starting to investigate effective mechanisms by which this attention can be directed, we are seeking to understand how the broad strokes of the attention investment model emerge. This could be helpful in exploring whether or not there

are places where this could be done more efficiently - but doing so might also generate information about the details of the Attention Investment framework; for example, does the misdirection of attention play a significance role in the way in which programmers perceive risk and reward?

The design of the Progger system also allows the possibility of integrating analyzers to further direct the programmers' attention. Such analyzers may be used to, for instance, facilitate a conversational-style interaction about considerations such as control flow. This may be explored in future work, however one key distinction to note between analyzers and the compiler is the inherent uncertainty of analysis results. Where a compiler error is indicative of a critical error that prevents the program from being executed, analysis tools are susceptible to false positives, and as such may direct the programmer's attention to an area of code that ultimately does not require fixing. False positives have previously been related by analysis tool users as one of the biggest factors in their low usage statistics, therefore the benefits of introducing features prone to false positives into the Progger system would need to be weighed carefully against the risks.

**Concluding Remarks** More widely our results indicate that in the context of programming in the face of errors, it is difficult to build a general conversational bridge between the programmers and compiler authors via the crude medium of error messages. However, whilst error messages can be problematic, the compiler directing the programmers attention to areas of the code, and the programmer being able to ask "*what were you looking at when you did x*" seem to be effective. It feels counter intuitive at first to abandon the richer communicative possibilities of error message text to focus only on the direction of attention, but it may prove a productive route for further exploration. Sometimes in conversations, it seems that less is more, especially when one of the participants (the compiler) does not really know what they are trying to say, and can not empathise effectively with the other.

## References

- [ASB19] M. Ahrens, K. Schneider, and M. Busch. "Attention in Software Maintenance: An Eye Tracking Study". In: *Proceedings of 6th International Workshop on Eye Movements in Programming (EMIP)*. 2019, pp. 2–9.
- [Bec+19] B. A. Becker et al. "Compiler error messages considered unhelpful: The landscape of text-based programming error message research". In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR)*. 2019, pp. 177–210.



- [Bed12] R. Bednarik. “Expertise-Dependent Visual Attention Strategies Develop over Time during Debugging with Multiple Code Representations”. In: *International Journal of Human-Computer Studies* 70.2 (2012), 143–155.
- [Ben+19] E. Beneteau et al. “Communication breakdowns between families and Alexa”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI ’19. 2019, pp. 1–13.
- [Bla02] A. F. Blackwell. “First steps in programming: A rationale for attention investment models”. In: *Proceedings of the IEEE 2002 Symposium on Human Centric Computing Languages and Environments*. IEEE. 2002, pp. 2–10.
- [Bus+15] T. Busjahn et al. “Eye Movements in Code Reading: Relaxing the Linear Order”. In: *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE. 2015, pp. 255–265.
- [Che+22] S. Cheng et al. “Collaborative eye tracking based code review through real-time shared gaze visualization”. In: *Frontiers of Computer Science* 16 (2022).
- [CSM21] L. Church, E. Söderberg, and A. T. McCabe. “Breaking down and making up-a lens for conversing with compilers”. In: *Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2021.
- [CSW02] M. Crosby, J. Scholtz, and S. Wiedenbeck. “The roles beacons play in comprehension for novice and expert programmers”. In: *Proceedings of the 14th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2002.
- [Cro05] N. Cross. *Engineering design methods: strategies for product design*. John Wiley & Sons, 2005.
- [Dub04] H. Dubberly. *How do you design?* Dubberly Design Office, 2004.
- [DP09] H. Dubberly and P. Pangaro. “What is conversation? How can we design for effective conversation”. In: *Interactions Magazine* 16.4 (2009), pp. 22–28.
- [EH07a] T. Ekman and G. Hedin. “The JastAdd system—modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.
- [EH07b] T. Ekman and G. Hedin. “The jastadd extensible java compiler”. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications (OOPSLA)*. 2007, pp. 1–18.

- [Gon+19] L. Gonçalves et al. “Measuring the Cognitive Load of Software Developers: A Systematic Mapping Study”. In: *Proceedings of 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 2019, pp. 42–52.
- [Hed00] G. Hedin. “Reference attributed grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.
- [Kat+09] L. C. L. Kats et al. “Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing”. In: *ACM SIGPLAN Notices* 44.10 (2009), pp. 445–464.
- [MSC21] A. T. McCabe, E. Söderberg, and L. Church. “Progger: Programming by Errors (Work In Progress)”. In: *Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group*. 2021.
- [Nor13] D. Norman. *The design of everyday things: Revised and expanded edition*. Basic books, 2013.
- [Pas76] G. Pask. “Conversation Theory”. In: *Applications in Education and Epistemology* (1976).
- [Pug81] S. Pugh. “Concept selection: a method that works”. In: *Proceedings of the International conference on Engineering Design*. 1981, pp. 497–506.
- [SH10] E. Söderberg and G. Hedin. “Automated selective caching for reference attribute grammars”. In: *Proceedings of the 3rd International Conference on Software Language Engineering (SLE)*. Springer. 2010, pp. 2–21.
- [Sto05] M.-A. Storey. “Theories, methods and tools in program comprehension: past, present and future”. In: *Proceedings of the 13th International Workshop on Program Comprehension (IWPC)*. 2005, pp. 181–191.



# INFLUENCING ATTENTION IN CODE READING: AN EYE-TRACKING STUDY

---

## Abstract

When interacting with other humans, we attempt to develop a shared understanding using various means. One such method is through our eyes: when someone is looking at something, we understand that their attention is focused on that object. In this work, we present the results of an eye-tracking study built upon the Progger tool, in which we used additional code highlighting in an attempt to influence the gaze behaviour of a human programmer, thereby focusing their attention. We found that though it is possible to draw attention towards areas of particular interest to the compiler, this has no apparent effect upon performance when confronted with a bug-finding code comprehension task. We conclude that although this strategy may be of use in the future when attempting to humanise the process of programming, further research is required to establish the efficacy of such interventions.

## 1 Introduction

In a series of recent research papers [MSC21; CSM21; MSC22], we have explored the idea of viewing the activity of programming as a "conversation" between two participants, namely the developer and their development environment. This "conversational lens" [CSM21], used as a "tool for thinking with"<sup>1</sup>, led to the development of a prototype tool, named Progger [MSC21].

---

<sup>1</sup>As coined at the industry panel during PPIG 2016 in Cambridge by Steven Clarke.

With Progger, we have explored the consequences of making visible the sequence of actions taken by a Java compiler in the lead up to a compiler error. This was an attempt to give the user a means of delving into the "thought process" of the compiler: much as in a human-to-human conversation a misunderstanding could be resolved by asking one participant to clearly and thoroughly explain what they are thinking about. This approach for humanising the interaction between developer and development environment centred around the usage of a linguistic-based strategy. Sections of code were highlighted and explicitly linked with descriptive text in the sidebar, which served as a summary of the computations made using attributes assigned to the code during compilation.

However, in our initial framing paper [CSM21], we also noted the use of so-called *side-channels* in human interactions. These side-channels are non-verbal cues which can greatly alter the meaning of speech, and can take on a variety of forms: body language, tone of voice, facial expressions, where a person is looking, and so forth.

Having built the Progger prototype, we conducted a pilot study [MSC22] with the aim of validating the design choices made in the development of the tool. Two interesting conclusions were drawn from this study: the additional descriptive text – the linguistic component – was found to be too esoteric for the target demographic of relative programming novices; and that the highlighting – the non-linguistic component – was of significant interest, drawing praise when it worked well and ire when it did not. In other words, Progger users were less interested in the somewhat abstract and complex thought process of the compiler, and much more interested in simply *where the compiler was looking*.

Being interested in where someone or something is looking is an innate characteristic in humans. The following of another person's gaze is a behaviour that has been found to start developing at a very young age [FBT07], and is of great significance in the development of social cognition. When two humans align their gaze, it signifies a "joint attention", indicating that both participants in the interaction are focused on the same thing. By aligning the "gaze" of a compiler with that of the developer, it may be possible to encourage the development of joint attention, with a focus on the most relevant areas of code. With this new insight an updated version of Progger was produced (Progger 2.0), stripping out extraneous text information and focusing solely on highlighting as a means of conveying information.

In this paper, we present the results of an eye-tracking study conducted using Progger 2.0, with the aim of understanding how showing where a program analysis tool is "looking" can influence the gaze and code reading behaviour of a human partner in the programming conversation, and help foster joint attention.

## 2 Background

Most visual tasks performed by humans are bottlenecked by the foveated nature of their visual system. Because the area of highest visual acuity of the human retina only spans a few degrees, visual tasks requiring resolving fine spatial detail often also require eye movements. As such, the study of eye movements has been fruitfully used to study the moment-to-moment unfolding of cognitive processes such as reading [Ray98] and visual search [HE96; Rao+02; Nie+19].

When viewing static visual scenes, such as pages of text or displays of programming code, humans predominantly exhibit two types of eye movements: fixations (periods during which the eye is still so that a relatively constant area of the visual scene is projected to the fovea to allow for fine visual processing) and saccades (rapid eye movements to bring gaze to the next area of interest in the scene) [Hes+18]. Eye trackers, devices that measure what a person looks at, are used to study such gaze behaviors (Holmqvist et al., 2011). In this study, like many before us [SSG15; OAH18; Kua+23], we make use of eye trackers to study how participants read programming code.

The act of program comprehension, when a person reads and attempts to understand unfamiliar code, has been the focus of multiple studies stretching across decades of research [CS90; Fei19]. For instance, a recent study [Bus+15] found that novices, as opposed to experts, have a tendency to read through code in a linear fashion, like how we would read a natural language text, and exhibited short average saccade length due to their eyes moving through the code from one line to the next. By comparison, experts were found to have a greater average saccade length and lower element coverage of the code, meaning that they focused on fewer lines of code and made larger jumps between lines as they read the code in a non-linear fashion.

By highlighting multiple non-consecutive lines of code, we hypothesised that analysis tools such as Progger 2.0 may have an affect on this phenomenon. Specifically, we speculate that by visualising the non-linear compiler gaze, we may encourage the user to spend a greater amount of time dwelling on highlighted lines, leading to the adoption of a similar gaze pattern and a more closely aligned focus of attention.

## 3 Method

With the aim of increasing our understanding of how showing the "attention" of a program analysis tool can influence human gaze and performance, we conducted an eye-tracking experiment. We broke down our objective into the following research questions:

**RQ<sub>1</sub>** Does the addition of compiler heatmap highlighting affect bug finding performance?

**RQ<sub>2</sub>** Does the addition of compiler heatmap highlighting affect gaze behavior when reading code?

### 3.1 Participants

In order to maintain consistency with the previous Progger study, as well as to reduce the number of independent variables, we decided to target novice programmers for the experiment. The participants were recruited from the pool of undergraduate computer science students at Lund University. An advertisement was initially made to students taking a course on agile software development, however the scope of the recruitment was later extended to include general advertising to the student cohort using Facebook groups. The only requirement for participation was the completion of at least one programming related class, and a total number of 15 students ultimately took part in the experiment. Of these participants, all were studying at an undergraduate level and none had any industrial experience outside of a summer internship. Participants were compensated for their participation in the form of a gift card for a cinema chain.

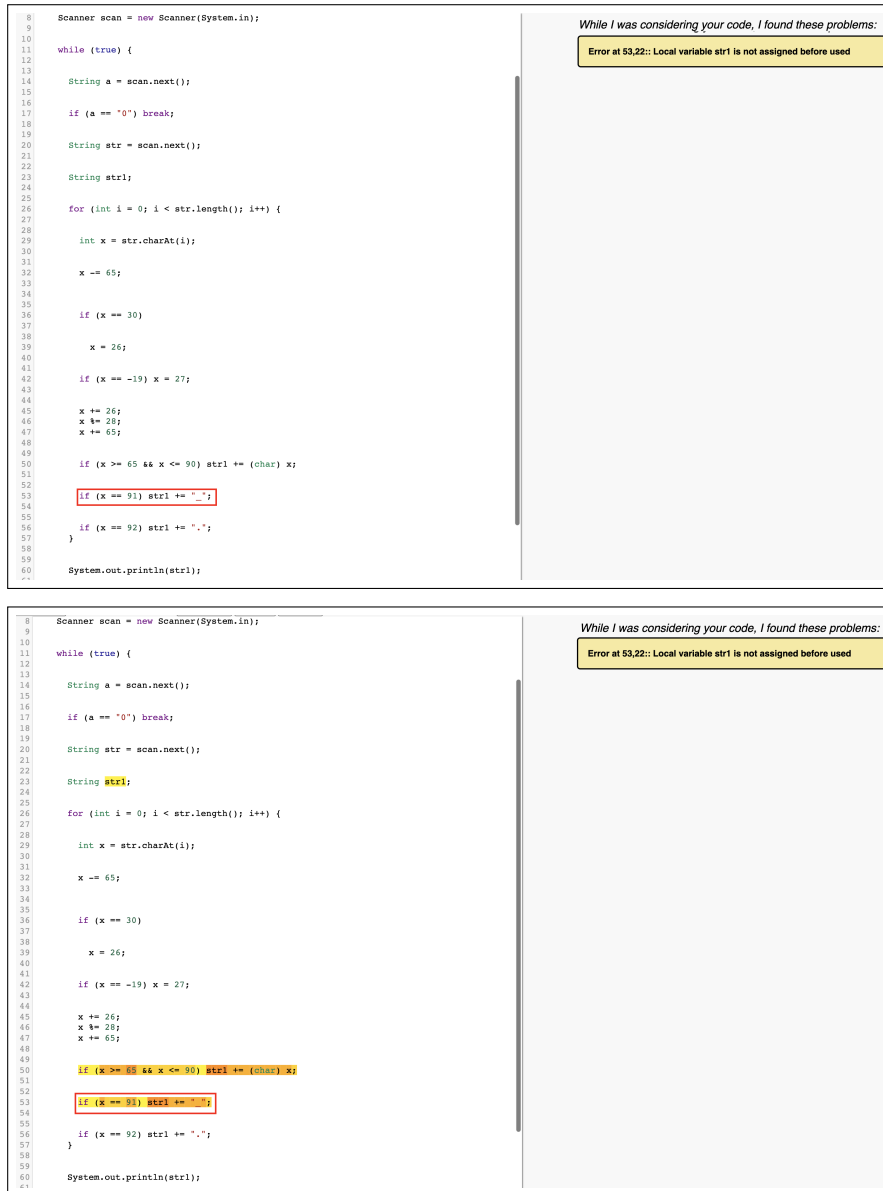
### 3.2 Stimuli

A set of eight stimuli was presented using Tobii Pro Lab, consisting of screenshots of small (8 to 17 lines) Java programs rendered within Progger, each containing at least one compiler error. A single error was identified with a red outline in the code, and the related error-message displayed in the side-bar.

Each stimulus contained either only a simple code snippet without highlighting and with only the error location indicated, or additionally contained highlighting provided by Progger indicating different lines of code which the compiler considered during computation of the error. An example stimulus showing both the non-highlighted and highlighted conditions is shown in Figure 1. Two sets of eight stimuli were created, sets A and B, each of which contained four non-highlighted code snippets and four with additional highlighting. The non-highlighted/highlighted screenshots were swapped between sets A and B. Of the 15 participants, 7 were shown stimulus set A and 8 were shown stimulus set B. The stimuli were presented in random order for each participant. It should be noted that extra whitespace was added between lines of code in order to achieve greater accuracy when determining exactly which line of code a fixation falls on.

### 3.3 Apparatus and Experimental Procedure

The experiment was conducted on-site at the Lund University Humanities Lab, using a Tobii Pro Spectrum that recorded gaze at 600 Hz. Each participant was seated at a booth which constricted their peripheral view, and viewed the stimuli from a viewing distance of approximately 63 cm on a 52.8 x 29.7 cm (47° x



**Figure 1:** An example of non-highlighted (top) versus highlighted (bottom) versions of a stimulus. In the stimulus, a compiler error has been thrown stating that a variable has not been assigned before use due to the initial assignment being dependent upon the results of a scanned input, which is indeterminate at compile time.



27°) computer screen (EIZO FlexScan EV2451, resolution 1920 x 1080 pixels) attached to the eye tracker while their head was placed on a chin- and forehead rest that was attached to the desk. The headrest and desk height were adjusted until the participant was comfortable. A five-point calibration procedure was executed followed by a four-point validation (mean accuracy 0.501 deg). An example of the experimental setup can be seen in Figure 2.



**Figure 2:** The experimental setup. The apparatus is contained within a booth, with the eye-tracker visible below the screen and a chin- and forehead rest in the foreground.

The eight code stimuli were presented during eight separate trials using Tobii Pro Lab. Each trial, the participants were asked to attempt to comprehend the code and determine both why the error had occurred, and how it might be fixed. Once they felt they had a good understanding of these questions, they were instructed to press any key in order to advance to a text-input screen where they were asked to summarise in their own thoughts why they believed the error had occurred, and how to resolve it. Once completed, they were able to move on to the next stimulus by pressing a key combination. In order to cater to the target demographic of novices, no advanced

language constructs or external libraries were used within the code samples. The whole experiment took approximately 15 to 50 minutes to complete, depending on participant.

### 3.4 Data Analysis

The experiment contained one independent variable: the state of the highlighting, either turned on or off for a given stimulus. Dependent variables included areas-of-interest (AOIs) analyses. To perform these analyses, for each of the experiment stimuli, areas-of-interest (AOIs) were created for each line of code and the error message. In the event of inconsequential lines of code (for example, a trailing `"}"` to close a block), these lines were grouped with the AOI defined for the preceding line. An example of a stimulus with AOIs defined is provided in Figure 3.

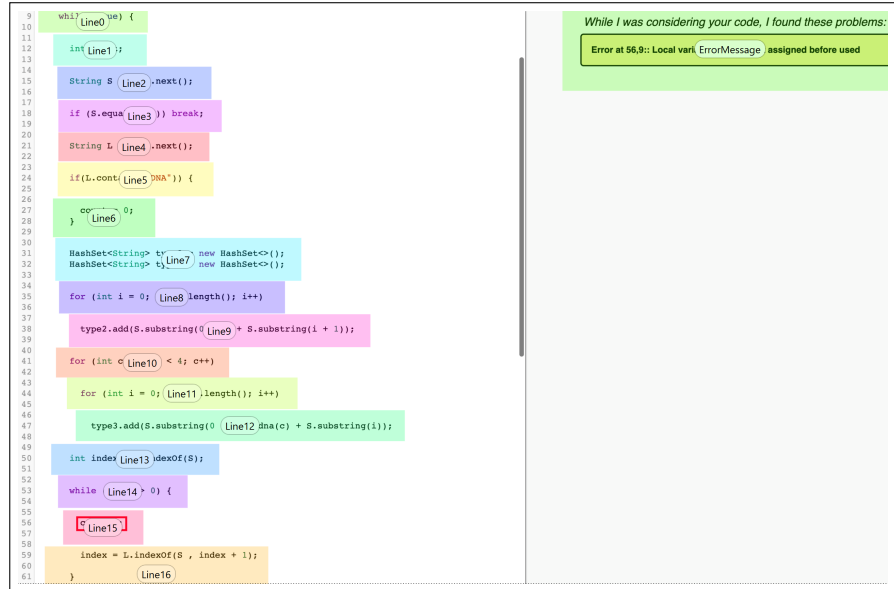
We used Tobii Pro Lab software [Tob23] to classify gaze into fixations using the default fixation filter with default settings, and then to annotate for each fixation whether it was in an AOI or not.

The output from Pro Lab was analyzed using a custom Java program. The program first reads in the data for a single stimulus and from this constructs a

"timeline" of AOI hits, corresponding to a list of consecutive AOI fixations. Some of the analyses listed below were performed using this timeline.

In total, six dependent variables are analyzed:

- **Time to completion:** how long a participant takes to solve a task, in seconds.
- **Correctness:** Whether the participant successfully solved a task, with a binary grading of either correct or incorrect. To compute correctness values, the documents where participants recorded their proposed solutions for each problem were analysed by the first author. Each solution was marked using a traffic light system, with red signifying an incorrect solution, yellow an incomplete solution or one where the participant demonstrated understanding but was unable to solve the issue, and green indicating a complete and correct solution. The third author then checked the proposed solution grading for correctness, and any disagreement was discussed until there was a consensus.. For statistical analysis purposes, the possible grades were then recoded to a binary format by coding incorrect or incomplete solutions as unsuccessful task solutions.
- **Hit-rate:** The percentage of AOI fixations that fell on lines that *would have been highlighted by Progger* in a given stimulus. This means that the hit-rate was calculated for the same lines regardless of whether the highlighted or non-highlighted version of a stimulus was presented. For example, in the stimulus shown in Figure 1 on line 23, the variable name is highlighted by Progger in the declaration statement: `String str1`. This AOI was therefore marked as an area of interest for both versions of the stimulus (regardless of the stimulus treatment).
- **Dwell duration:** The average amount of time the participant looked at an area of interest before moving their gaze to another part of the screen, in milliseconds. Like for hit-rate, for the dwell duration calculation only highlighted lines or lines that would have been highlighted by Progger are considered. Dwell times were computed by summing together the duration of consecutive fixations that fell in the same AOI.
- **Saccade length:** The average distance between gaze fixations while reading the code, in degrees. Specifically, the saccade classification output provided by Tobii Pro Lab was used, and the distance between the two fixations that are adjacent to each saccade computed.
- **Linearity:** The percentage of gaze movements corresponding to a linear forward change. Specifically, we counted consecutive AOI fixation pairs that represented a *single-step forward-progression* in corresponding lines. For example, consecutive AOI fixations of lines 5 and 6 would constitute



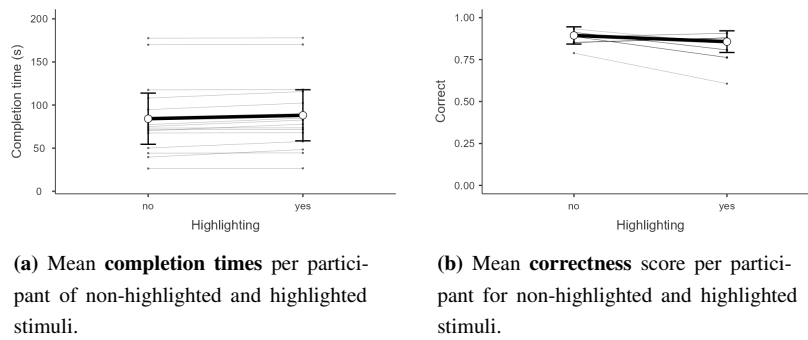
**Figure 3:** The areas-of-interest for a stimulus, as defined in Tobii Pro Lab.

a linear change and thus contribute to this metric. Examples of pairs that would not contribute are fixations on lines 5 and 7 (multiple-step) or lines 6 and 5 (backwards-progression).

The data were analyzed in Jamovi version 2.3 [pro22; R C21] with the GAMLj module [Gal19]. Linear mixed effect modelling was performed with highlighting as a fixed effect and participant and stimulus as random effects for five out of the six dependent variables. Correctness was instead analyzed using a generalized mixed model to perform a logistic regression using a logit link function. As for the other analyses, highlighting was specified as a fixed effect and participant and stimulus as random effects. Data plots were also created using Jamovi and show data for individual participants along with the mean across participants. Error bars denote standard errors of the mean.

### 3.5 Threats to validity

The primary threats to the validity of this study are the low **sample sizes**. To ensure the greatest accuracy of the eye-tracking equipment, we found it necessary to invite participants on-site to the Lund University Humanities Lab, which may have been a deterrent for some people as they were impelled to go out of their way to contribute. We attempted to offset this by offering an incentive, however the



**Figure 4:** Charts detailing the completion time and correctness metrics.

total number of participants was ultimately quite low at 15. To improve the internal validity of the study, it would be desirable to recruit a larger pool of participants.

Similarly, the relatively low **stimuli number** could have had an effect on the data. This led to several participants completing the tasks very quickly, however some participants also used the entire 1 hour allotted to them. For future studies, it may be desirable to increase the number of stimuli, although with the caveat that this could impact participation numbers.

**Sample selection** may also have affected the internal validity of the study, as only novices were recruited for participation. This led to some difficulties for specific tasks, and may have impacted the validity of the correctness metric in particular.

## 4 Results

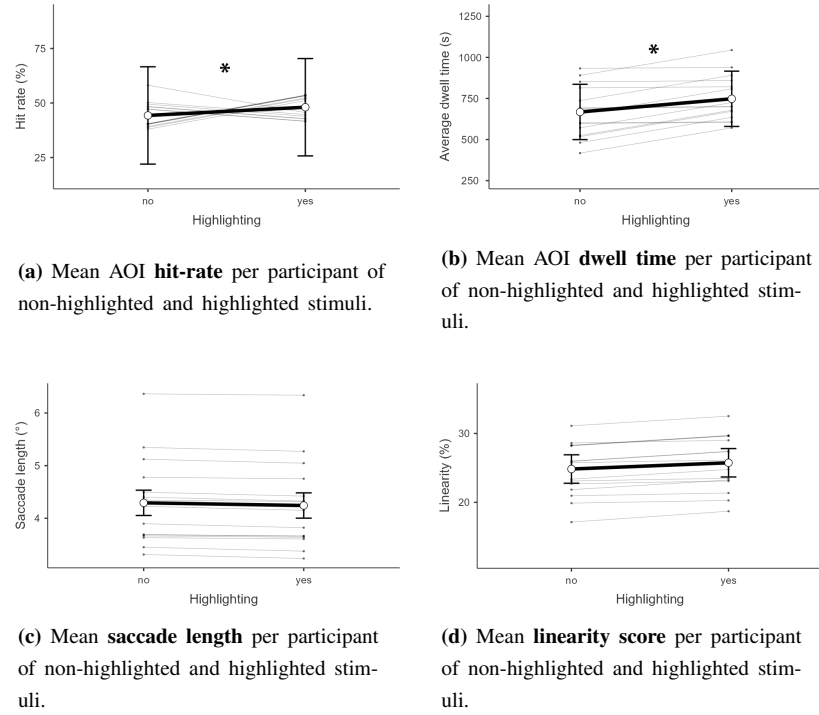
Here we present the results broken down into performance and gaze metrics on AOIs.

### 4.1 Time to complete tasks

In order to answer RQ<sub>1</sub>, the time taken to solve each trial was analysed. As can be seen in Figure 4 (a), highlighting had no significant effect on the completion time ( $F(1, 96.5) = 0.258, p = 0.613$ ).

### 4.2 Correctness

Across all participants, 101 of 120 trials (84%) yielded a correct solution. Statistical analysis was again performed to investigate whether the addition of highlight-



**Figure 5:** Charts detailing the collected eye-tracking data. An asterisk (\*) indicates a statistically significant effect of highlighting change.

ing had an effect, as can be seen in Figure 4 (b). The influence of highlighting on mean correctness was found to be insignificant ( $\chi^2(1.00) = 0.401, p = 0.527$ ).

### 4.3 Highlighted line hit-rate

The mean hit-rates across all participants were then calculated and analysed, as seen in Figure 5 (a). Hit-rate was larger for highlighted stimuli than non-highlighted stimuli ( $F(1, 95.8) = 10.6, p = 0.002$ ), indicating that highlighted lines of code are looked at more.

### 4.4 Dwell duration

The results of dwell time analysis can be seen in Figure 5 (b), and, consistent with the hit rate metric, exhibit an increase of 80 ms in average dwell duration between non-highlighted and highlighted versions of the stimuli ( $F(1, 96.2) = 4.53, p =$

0.036). This indicates that participants spent more time looking at lines of code that the compiler considered to be of interest when the highlighted versions of the stimuli were presented to them.

#### 4.5 Saccade length

No significant differences in saccade length were found between highlighted and non-highlighted stimuli ( $F(1, 97.7) = 0.184, p = 0.669$ ), see Figure 5 (c).

#### 4.6 Reading linearity

As can be seen from Figure 5 (d), there was no significant difference in linearity scores between the non-highlighted and highlighted conditions ( $F(1, 96.3) = 0.490, p = 0.486$ ). Together with the saccade length metric, this finding suggests that the way in which participants scanned the stimuli did not differ depending on availability of highlighting.

### 5 Discussion

Of the six dependent variables which were analysed, we used the first two (completion time and correctness) to answer RQ<sub>1</sub> (compiler heatmap highlighting effect on bug finding performance). We found that the addition of compiler heatmap highlighting did not have an effect on both the completion time ( $p = 0.613$ ) and the correctness ( $p = 0.527$ ).

We used the analysis of dependent variables three to six (hit-rate, dwell duration, saccade length, linearity) to answer RQ<sub>2</sub> (compiler heatmap highlighting effect on gaze behaviour when reading code). Of these variables, the closely related metrics of saccade length and linearity displayed no significant difference between the two highlighting conditions. This may suggest that, in the act of program comprehension, novice programmers exhibit similar patterns of gaze movements when reading unfamiliar code, regardless of additional compiler heatmap highlighting.

By contrast, the metrics relating to where (AOI hits) and for how long (dwell duration) the participants looked show a marked difference. From the data, we see that by adding highlighting to a line, novice programmers tend to look more often, and for longer periods of time, at that line. Despite this, as related in the discussion on RQ<sub>1</sub>, this difference in gaze behaviour has no significant effect on either time to complete a task or correctness of the solution when presented with a code comprehension problem.

The effect of higher hit-rate and dwell duration ultimately warrants further investigation, as there were several confounding factors which may have influenced the results. It may be that the experimental setup being focused on comprehension was not conducive to the highlighting aiding the participants: a desire to

understand the code accurately may have led participants to reading in a very methodical way. We believe that in the future it would be worth studying the effect of compiler heatmap highlighting in different contexts, such as when examining control flow, or when investigating a bug in an already familiar code base. The relatively low number of participants may also have affected our ability to detect effects of highlighting, and it may be worth attempting to re-run the study with a larger number of subjects.

The low level of experience across all participants may also have fed into the results due to their lack of familiarity with certain features of the Java language. For instance, the stimulus which received most incorrect solutions (6 out of 15 responses, or 40%) was centred around the use of the `final` keyword. Some participants were unsure how this would affect the mutability of the relevant variable, and thus offered incorrect solutions.

Despite these factors, the experiment led to interesting findings when considering the concept of joint attention. The method in which the heatmap highlighting is computed (as described in past papers, [MSC21; MSC22]) is based on the lines which are considered by the compiler in the code analysis resulting in a found error. When no highlighting exists on a line, the compiler did not consider it to be of importance. In contrast, the darkness of the highlighting on a given line or term is related to the number of times the compiler "gaze" passed over this area of code. The fact that participants looked at highlighted sections more often, and for longer periods, shows that this visualisation of compiler attention had a notable effect on user attention.

One of the main motivations in developing the conversational lens for analysing interactions with a programming environment was to draw upon human characteristics to make the process more natural. We believe that in moving towards more natural, instinctive methods of communication, many of the abstractions of human-computer interaction, for example hiding a complex compilation process behind a simple error message, can be made more clear. In this study we have shown that, although no effects were found on performance in this experimental setup, it is indeed possible to use interaction design to encourage the very human phenomenon of joint attention with a computer.

## 6 Acknowledgements

The authors gratefully acknowledge Lund University Humanities Lab for providing the facilities for the experiment, and would like to thank Prof. Martin Höst for early feedback on the experiment design. This work has been partially supported by the Swedish Foundation for Strategic Research (grant no. FFL18-0231), the Swedish Research Council (grant no. 2019-05658), ELLIIT - the Swedish Strategic Research Area in IT and Mobile Communications, and the Wallenberg

AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

## References

- [Bus+15] T. Busjahn et al. “Eye Movements in Code Reading: Relaxing the Linear Order”. In: *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE. 2015, pp. 255–265.
- [CSM21] L. Church, E. Söderberg, and A. T. McCabe. “Breaking down and making up-a lens for conversing with compilers”. In: *Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group (PPIG)*. 2021.
- [CS90] M.E. Crosby and J. Stelovsky. “How do we read algorithms? A case study”. In: *Computer* 23.1 (1990), pp. 25–35.
- [Fei19] D. G. Fietelson. “Eye Tracking and Program Comprehension”. In: *Proceedings of the 2019 IEEE/ACM 6th International Workshop on Eye Movements in Programming (EMIP)*. 2019, pp. 1–1.
- [FBT07] A. Frischen, A. P. Balyiss, and S. P. Tipper. “Gaze Cueing of Attention: visual attention, social cognition, and individual differences”. In: *Psychological bulletin* 133.4 (2007), pp. 694–724.
- [Gal19] Galluci, M. *GAMLj: General analyses for linear models*. Version 2.6. 2019.
- [Hes+18] R. S. Hessels et al. “Is the eye-movement field confused about fixations and saccades? A survey among 124 researchers”. In: *Royal Society Open Science* 5 (8 2018).
- [HE96] I. T. C. Hooge and C. J. Erkelens. “Control of fixation duration in a simple search task”. In: *Perception & Psychophysics* 58.7 (1996), pp. 969–976.
- [Kua+23] P. Kuang et al. “Towards Gaze-Assisted Developer Tools”. In: *Proceedings of the 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE. 2023.
- [MSC21] A. T. McCabe, E. Söderberg, and L. Church. “Progger: Programming by Errors (Work In Progress).” In: *Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group*. 2021.
- [MSC22] A. T. McCabe, E. Söderberg, and L. Church. “Visual Cues in Compiler Conversations”. In: *Proceedings of the 33rd Annual Workshop of the Psychology of Programming Interest Group Annual Workshop (PPIG)*. 2022.



- [Nie+19] D. C. Niehorster et al. “Searching with and against each other: Spatiotemporal coordination of visual search behavior in collaborative and competitive settings.” In: *Atten Percept Psychophys* 81 (2019), pp. 666–683.
- [OAH18] U. Obaidallah, M. Al Haek, and P. C.-H. Cheng. “A Survey on the Usage of Eye-Tracking in Computer Programming”. In: *ACM Computing Surveys* 51.1 (2018).
- [R C21] R Core Team. *R: A Language and environment for statistical computing*. Version 4.1. 2021.
- [Rao+02] R. P. N. Rao et al. “Eye movements in iconic visual search”. In: *Vision Research* 42.11 (2002), pp. 1447–1463.
- [Ray98] K. Rayner. “Eye movements in reading and information processing: 20 years of research”. In: *Psychological Bulletin* 124.3 (1998), pp. 372–422.
- [SSG15] Z. Sharafi, Z. Soh, and Y.-G. Guéhéneuc. “A systematic literature review on the usage of eye-tracking in software engineering”. In: *Information and Software Technology* 67 (2015), pp. 79–107.
- [Tob23] Tobii AB. *Tobii Pro Lab*. Version 1.217. 2023.
- [pro22] the jamovi project. *jamovi*. Version 2.3. 2022.