Visual Cues in Compiler Conversations

Alan T. McCabe
Lund University
alan.mccabe@cs.lth.se

Emma Söderberg
Lund University
emma.soderberg@cs.lth.se

Luke Church
University of Cambridge
luke@church.name

Peng Kuang
Lund University
peng.kuang@cs.lth.se

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 (Beneteau et al., 2019), and may even mislead the reader into looking at the wrong sections of code altogether (Becker et al., 2019; Kats, de Jonge, Nilsson-Nyman, & Visser, 2009). This "feedback" is not only limited in form, but is also delivered in a purely binary format - an error occurs, or it does not.

In our previous work, "Breaking down and making up - a lens for conversing with compilers" (Church, Söderberg, & McCabe, 2021), 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 (Dubberly & Pangaro, 2009) and Pask (Pask, 1976), 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 (Dubberly & Pangaro, 2009), 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 (McCabe, Söderberg, & Church, 2021), 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.

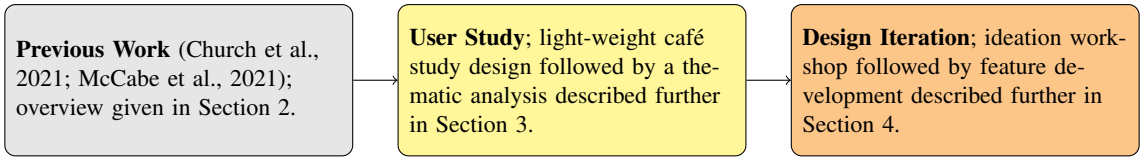


*Figure 1 – **Overview** of the work presented in this paper and its disposition.*

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) (Gonçales, Farias, da Silva, & Fessler, 2019). 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., (Busjahn et al., 2015; Storey, 2005); 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. (Crosby, Scholtz, & Wiedenbeck, 2002) 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 (Bednarik, 2012) 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. (Ahrens, Schneider, & Busch, 2019) 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. (Cheng, Wang, Shen, Chen, & Dey, 2022). 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 (Pask, 1976). 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 (Beneteau et al., 2019). 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 (Dubberly & Pangaro, 2009).

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 (Church et al., 2021). 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 (McCabe et al., 2021). This tool, Progger, consists of a Dart[1] front-end communicating via a REST API with a Java compilation server. The compilation service contains a small extension to the extendable Java compiler ExtendJ[2] (Ekman & Hedin, 2007a), which itself is based on the JastAdd[3] meta-compilation system (Ekman & Hedin, 2007b). 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 (Hedin, 2000). An explanation of this formalism is beyond the scope of this paper, however a detailed description of reference attribute grammars and their significant within the Progger system may be found in our previous work, "Progger: Programming by Errors (Work In Progress)" (McCabe et al., 2021).

By use the tracing system inherent in JastAdd (Söderberg & Hedin, 2010), 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 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.

[1] The Dart programming language, `https://dart.dev/`.

[2] The Extensible Java Compiler ExtendJ, `https://extendj.org`.

[3] The meta-compilation system JastAdd, `https://jastadd.org`.

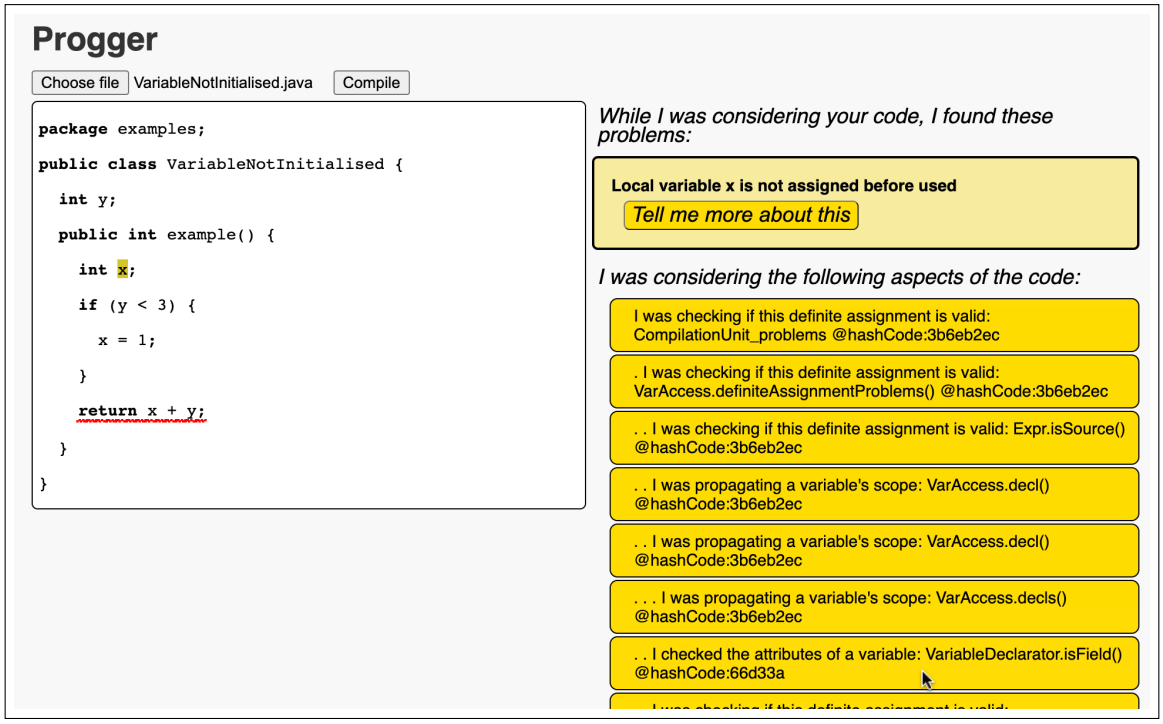


Figure 2 – Screenshot of Progger version 1.0. The left side shows a small Java code snippet with an error in it pointed out with a red squiggly line. The right side shows the error with a "Tell me more about this" button which has been clicked here to expand an attribute trace tree shown in the bottom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

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 it's 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[4] of solutions to Kattis[5] programs, with a selection of errors manually inserted into the code by the first author. The error are representative of a small set of semantic error patterns, selected to provide a instances where the prototype was found to be particularly helpful in

[4]Provided with permission by Pedro Contipelli: https://github.com/PedroContipelli/Kattis, visited at commit 30884ba
[5]Kattis problem archive: https://open.kattis.com/

Figure 3 – The user study booth.

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 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"*),

	Participants												
Code	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			

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).

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 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

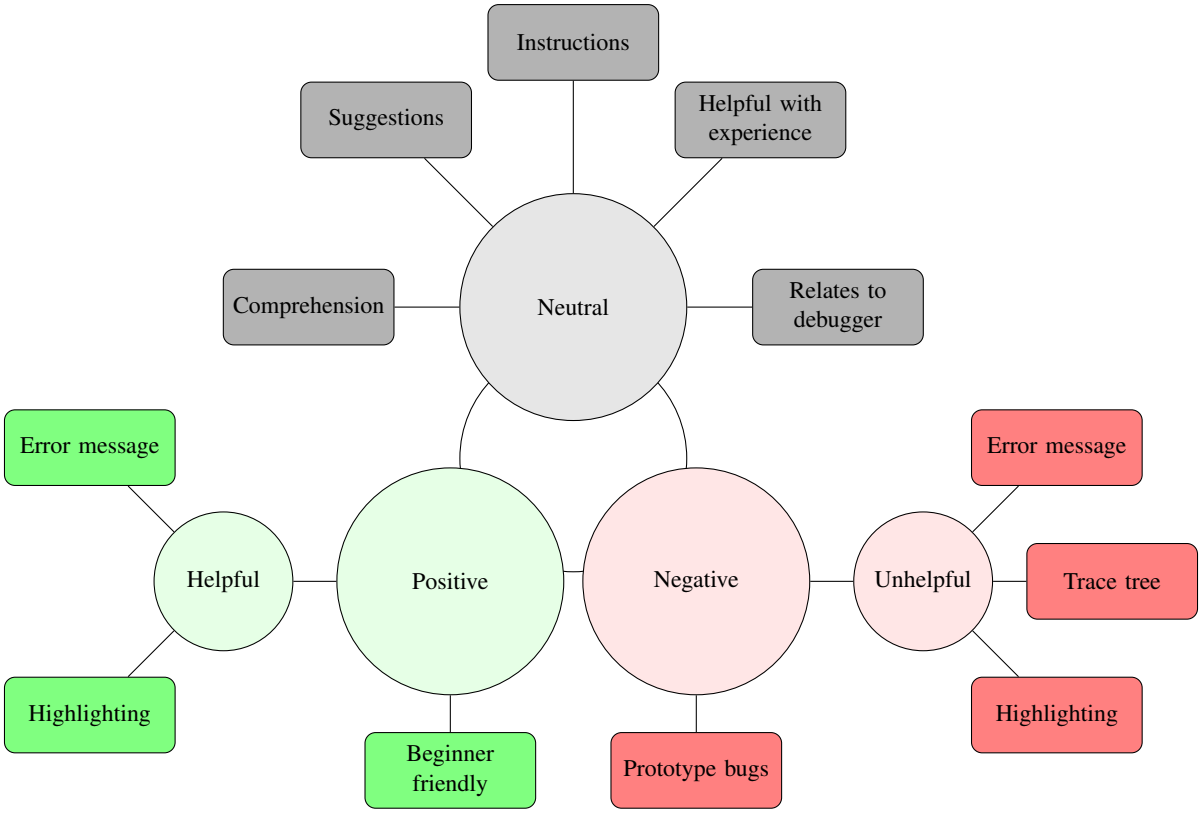


*Figure 4 – Identified **themes**, represented by circles, and associated **codes**, represented by boxes.*

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. (Crosby et al., 2002). 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 gulf of evaluation (Norman, 2013).

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 (Cross, 2005; Dubberly, 2004; Pugh, 1981). 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.

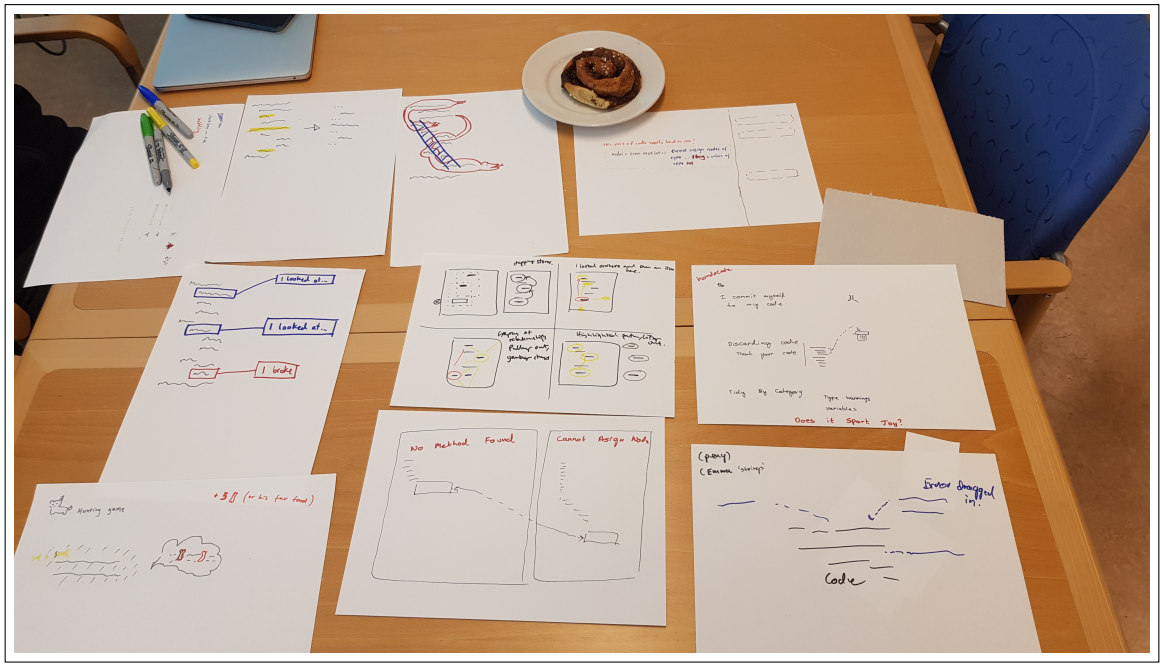


Figure 5 – A selection of design concepts as produced by the ideation workshop.

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.

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.

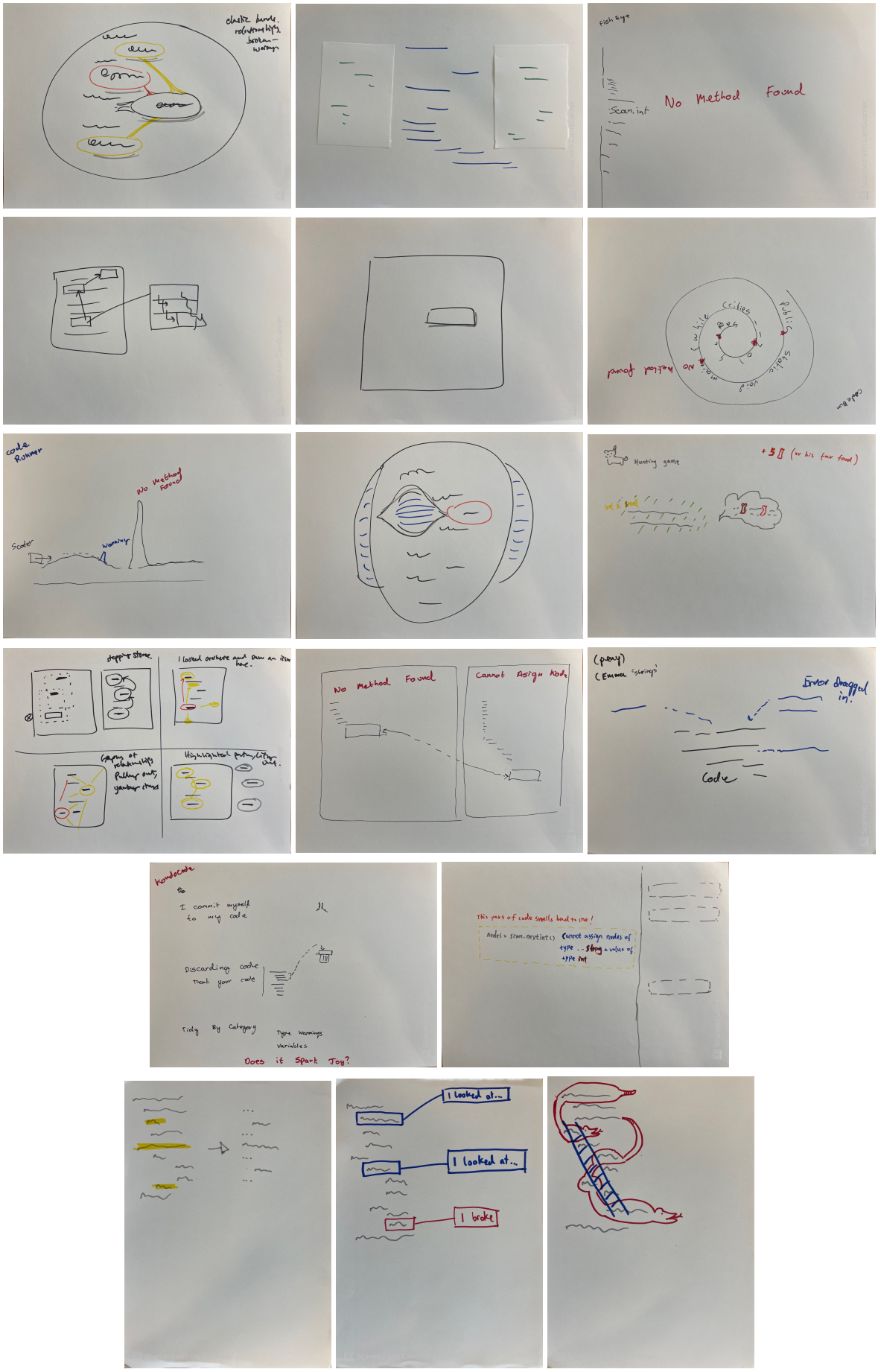
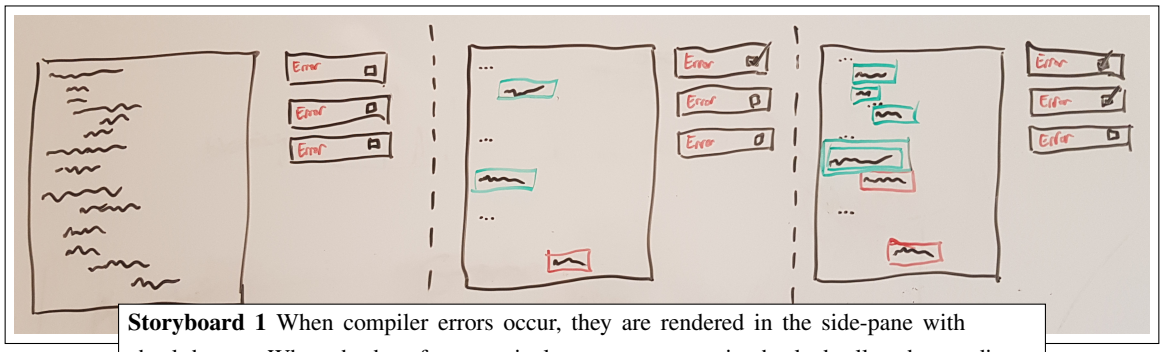
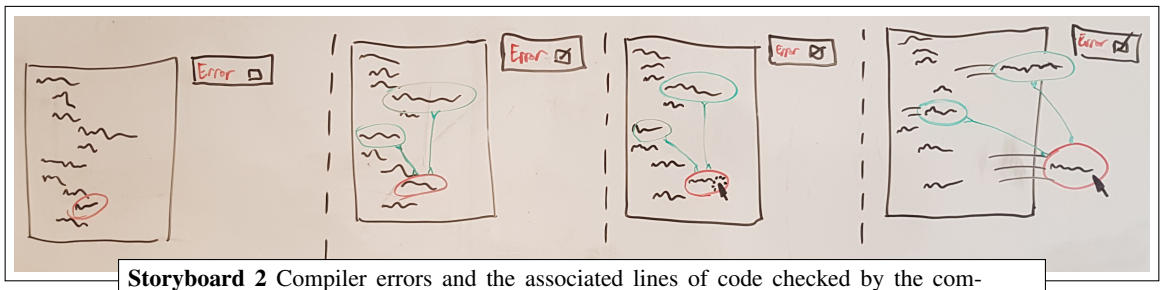


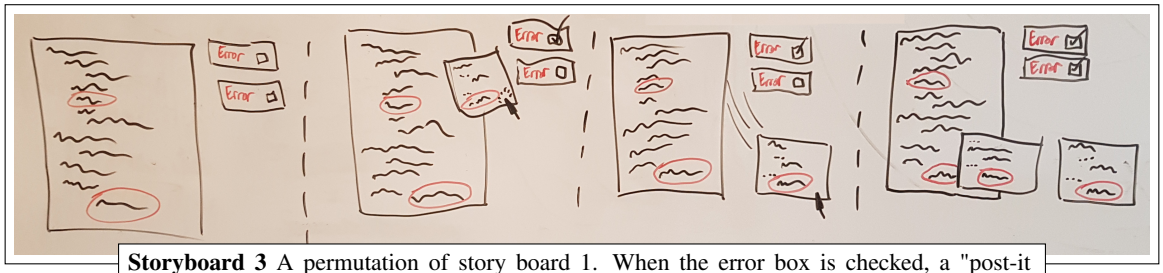
Figure 6 – A collage of the ideas conceptualised during the ideation session.



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.*

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.

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 (McCabe et al., 2021). 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. (Ahrens et al., 2019) 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. (Cheng et al., 2022), 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 (Blackwell, 2002) 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.

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.

6. References

Ahrens, M., Schneider, K., & Busch, M. (2019). Attention in software maintenance: An eye tracking study. , 2-9. doi: 10.1109/EMIP.2019.00009

Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., ... others (2019). 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* (pp. 177–210).

Bednarik, R. (2012, feb). Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *Int. J. Hum.-Comput. Stud.*, *70*(2), 143–155. doi: 10.1016/j.ijhcs.2011.09.003

Beneteau, E., Richards, O. K., Zhang, M., Kientz, J. A., Yip, J., & Hiniker, A. (2019). Communication breakdowns between families and alexa. In *Proceedings of the 2019 CHI conference on human factors in computing systems* (pp. 1–13).

Blackwell, A. F. (2002). First steps in programming: A rationale for attention investment models. In *Proceedings ieee 2002 symposia on human centric computing languages and environments* (pp. 2–10).

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., ... Tamm, S. (2015). Eye movements in code reading: Relaxing the linear order. In *2015 ieee 23rd international conference on program comprehension* (p. 255-265). doi: 10.1109/ICPC.2015.36

Cheng, S., Wang, J., Shen, X., Chen, Y., & Dey, A. (2022, 06). Collaborative eye tracking based code review through real-time shared gaze visualization. *Frontiers of Computer Science*, *16*. doi: 10.1007/s11704-020-0422-1

Church, L., Söderberg, E., & McCabe, A. (2021). Breaking down and making up-a lens for conversing with compilers. In *Psychology of programming interest group annual workshop 2021.*

Crosby, M., Scholtz, J., & Wiedenbeck, S. (2002, 07). The roles beacons play in comprehension for novice and expert programmers. In *Psychology of programming interest group annual workshop 2002.*

Cross, N. (2005). *Engineering design methods: strategies for product design*. John Wiley & Sons.

Dubberly, H. (2004). How do you design. *A compendium of models*, *10*.

Dubberly, H., & Pangaro, P. (2009). What is conversation? how can we design for effective conversation. *Interactions Magazine*, *16*(4), 22–28.

Ekman, T., & Hedin, G. (2007a). The jastadd extensible java compiler. In *Proceedings of the 22nd annual acm sigplan conference on object-oriented programming systems, languages and applications* (pp. 1–18).

Ekman, T., & Hedin, G. (2007b). The jastadd system—modular extensible compiler construction. *Science of Computer Programming*, *69*(1-3), 14–26.

Gonçales, L., Farias, K., da Silva, B., & Fessler, J. (2019). Measuring the cognitive load of software developers: A systematic mapping study. In *2019 ieee/acm 27th international conference on program comprehension (icpc)* (p. 42-52). doi: 10.1109/ICPC.2019.00018

Hedin, G. (2000). Reference attributed grammars. *Informatica (Slovenia)*, *24*(3), 301–317.

Kats, L. C., de Jonge, M., Nilsson-Nyman, E., & Visser, E. (2009). Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-lr parsing. *ACM SIGPLAN Notices*, *44*(10), 445–464.

McCabe, A. T., Söderberg, E., & Church, L. (2021). Progger: Programming by errors (work in progress). In *Psychology of programming interest group annual workshop 2021*.

Norman, D. (2013). *The design of everyday things: Revised and expanded edition*. Basic books.

Pask, G. (1976). Conversation theory. *Applications in Education and Epistemology*.

Pugh, S. (1981). Concept selection: a method that works. In *Proceedings of the international conference on engineering design* (pp. 497–506).

Söderberg, E., & Hedin, G. (2010). Automated selective caching for reference attribute grammars. In *International conference on software language engineering* (pp. 2–21).

Storey, M.-A. (2005). Theories, methods and tools in program comprehension: past, present and future. In *13th international workshop on program comprehension (iwpc'05)* (p. 181-191). doi: 10.1109/WPC.2005.38