Breaking down and making up - a lens for conversing with compilers

Luke Church
Lund University / University of Cambridge
luke@church.name

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

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

Abstract
This paper proposes a 'tool for thinking with'[1]: 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.

Keywords
Conversations, Developer Tools, Usability, Communication breakdown, Conversational repair

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 (Cox & McLean, 2012), whilst our (Sadowski et al, 2018) and others' previous work (Bacchelli & Bird, 2013), 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 (Church, 2018), or the computer that just says 'no' (or don't know) (A. Blackwell et al., 2018), 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 (Johnson et al., 2013), to problems in CS Education (Becker et al., 2019), 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 centers the interactions onto the terms of the computer, not the people doing the development.

[1] A phrase coined by Steven Clarke at the industry panel, PPIG 2016, Cambridge

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[2]. Whilst the notion of considering interaction as a conversation has been considered earlier by, for instance, (Dubberly & Pangaro, 2009) building on the work of (Pask, 1976), 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 (Pask, 1976), conversation analysis (Sacks et al., 1978), interaction design and conversations (Dubberly & Pangaro, 2009), conversational alignment (Henderson & Harris, 2011), communication breakdown (Beneteau et al., 2019) and properties of good conversions (Clark et al., 2019), 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, 2009) 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., 1978). This turn-taking model includes turn-constructional 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 (Tanimoto, 1990), and the broader live programming communities have studied the intertemporal nature of the interaction between developers and their tools about the

[2] 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

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 (Clark et al., 2019). 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* (Searle, 1996), that is between the people in the conversation, and that a number of mechanisms are used to determine *conversational alignment* (Henderson & Harris, 2011) - whether they are "on the same page". Utilization of conversations to create a shared understanding ("*meaning making*") and to reach agreement (Dubberly & Pangaro, 2009) is central in conversation theory (Pask, 1976), 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 (Clark et al., 2019).

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* (Rogers & Farson, 2015) 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 (Sacks et al., 1978). In this repair, they enter what may be referred to as a meta-conversation (Dubberly & Pangaro, 2009), 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., 2019), 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 (Johnson et al., 2013) 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 (Imtiaz et al., 2019) 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., 2016) 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 (Sacks et al., 1978), explicability is closely related to the construction of a shared understanding (Pask, 1976) and meaning making (Dubberly & Pangaro, 2009), where it helps to bring about conversational alignment (Henderson & Harris, 2011).

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., 2019) 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., 2013).

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 (Novick et al., 1996).

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 (Lohmeier, 2016), 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 (Blackwell & Green, 2000) 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 (Green & Petre, 1996) and The Patterns of User eXperience (Blackwell, 2015) , 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.

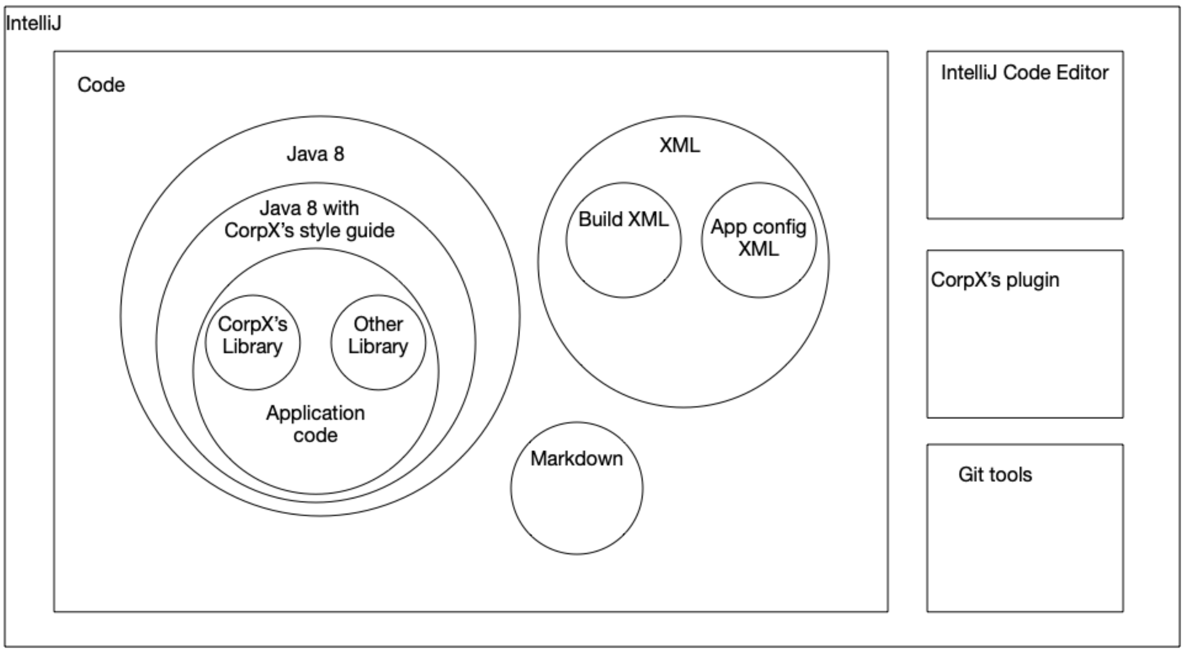


Figure 1. A schematic outline on the number of different languages, both notational and interactional that are involved in a notional commercial software development context (called CorpX).

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[3], 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 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 (Green, 1990) ecosystem where any improvement has to be supported across a number of different tools in order to achieve practical usability within an

[3] 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

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 (Church et al., 2010). 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 (Oulasvirta et al., 2018)

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 (Blackwell, 2000) 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 (Basman et al., 2016) 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 (Mărăşoiu et al., 2015), 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.

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.

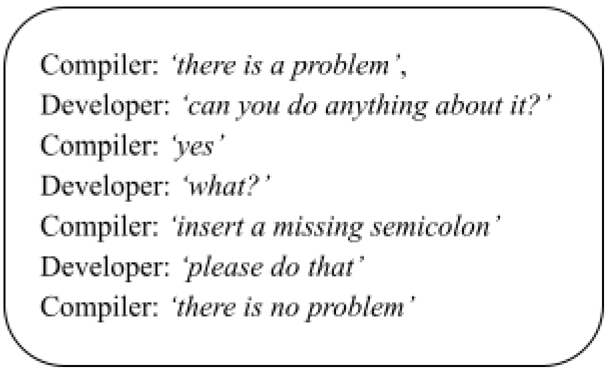

Compiler: *'there is a problem'*,
Developer: *'can you do anything about it?'*
Compiler: *'yes'*
Developer: *'what?'*
Compiler: *'insert a missing semicolon'*
Developer: *'please do that'*
Compiler: *'there is no problem'*

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

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.

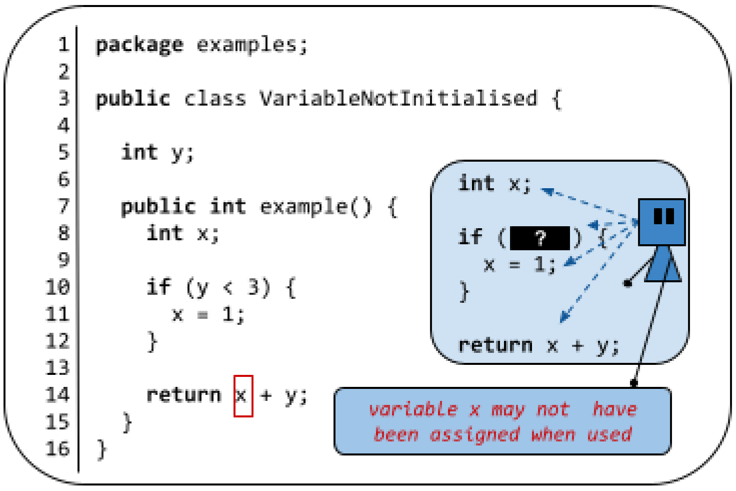
There is no way for a developer to ask the compiler basic questions like "why is that a problem?", or "what were you doing when you had this problem?", or "can you show me another example of this problem?". These requests would be part of a normal conversation with an experienced developer in understanding 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[4]

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, 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?

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 (Mori et al., 2012). Instead we aim to implicitly support the conversational nature of the interaction outlined in the properties above.



```
package examples;

public class VariableNotInitialised {

  int y;

  public int example() {
    int x;

    if (y < 3) {
      x = 1;
    }

    return x + y;
  }
}
```

```
int x;

if ( ? ) {
  x = 1;
}

return x + y;
```

```
variable x may not have
been assigned when used
```

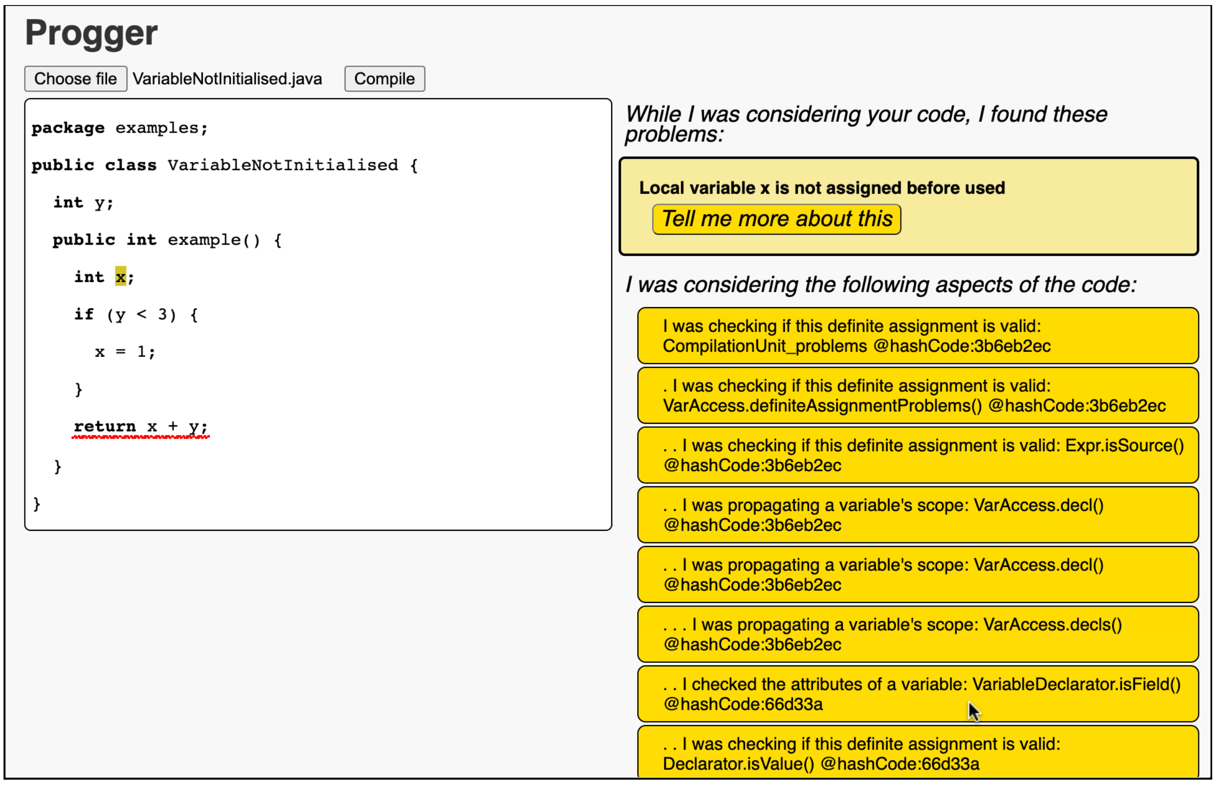
Figure 3. Java code example with the compiler and its view illustrated with blue figures.

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

[4] 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.

developer. To solve this 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 (Ekman & Hedin, 2007), a compiler built upon the JastAdd meta-compilation system (Hedin & Magnusson, 2003). JastAdd is an implementation of the *referenced attribute grammars* formalism, as described in (Hedin, 2000), 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.



```
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

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

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 (Church and Söderberg 2019) 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.

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

Bacchelli, A. & Bird, C. (2013). *Expectations, outcomes, and challenges of modern code review.* In proceedings of the 35th International Conference on Software Engineering (ICSE), pp. 712-721, IEEE.

Basman, A., Church, L., Klokmose, C., & Clark, C. B. D. (2016). *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).

Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., Kamil, A., Karkare, A., McDonald, C., Osera, P., Pearce, J. L., & Prather, J. (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, ACM.

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, ACM.

Blackwell, A. (2015). *Patterns of User Experience in Performance Programming.* In proceedings of the 1st International Conference on Live Coding (ICLC), pp. 12-22.

Blackwell, A., Church, L., Jones, M., Jones, R., Mahmoudi, M., Mărășoiu, M., Meakins, S., Nauck, D., Prince K., Semrov, A., Simpson, A., Spott, M., Vuyksteke, A. & Wang, X. (2018). *Computer Says 'don't Know' - Interacting Visually with Incomplete AI Models.* Talk at DTSHPS workshop, Co-located with VLHCC.

Blackwell, A. F. (2000). Dealing with New Cognitive Dimensions. Position paper prepared for the Workshop on Cognitive Dimensions, University of Hertfordshire, United Kingdom.

Blackwell, A. F. & Green, T. (2000). *A Cognitive Dimensions Questionnaire Optimised for Users.* In proceedings of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG).

Church, L. (2018). *Critique of 'Lector in Codigo or Role of the Reader'".* In the Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, ACM.

Church, L., Nash, C., & Blackwell, A. F. (2010). *Liveness in Notation Use: From Music to Programming.* In proceedings of the 21th Annual Workshop of the Psychology of Programming Interest Group (PPIG).

Church, L., & Söderberg, E. (2019). *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).

Cox, G., & McLean, A. (2012). *Speaking Code*. MIT Press.

Clark, L., Pantidi, N., Cooney, O., Doyle, P., Garaialde, D., Edwards, J., Spillane, B., Gilmartin, E., Murad, C., Munteanu, C., Wade, V., & Cowan, B. R. (2019). *What Makes a Good Conversation? Challenges in Designing Truly Conversational Agents.* In proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, ACM.

Dubberly H., & Pangaro, P. (2009). *What is conversation? How can we design for effective conversation?* Interactions Magazine, XVI(4):22-28.

Ekman, T., & Hedin, E. (2007). *The JastAdd Extensible Java Compiler.* SIGPLAN Notices 42(10):1-18.

Fors, N., Söderberg, E., & Hedin, G. (2020). *Principles and Patterns of JastAdd-style Reference Attribute Grammars.* In proceedings of the 13th International Conference on Software Language Engineering, ACM.

Green, T. R., G. (1990). *The Cognitive Dimensions of Viscosity: A Sticky Problem for HCI.* In proceedings of the IFIP TC13 Third International Conference on Human-Computer Interaction (INTERACT), pp. 79-86, North-Holland Publishing Co..

Green, T. R. G., & Petre, M. (1996). *Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework.* Journal of Visual Languages & Computing, 7(2):131-174, Elsevier.

Hedin, G. (2000). *Reference Attributed Grammars*. Informatica, 24(3):301–317.

Hedin, G., & Magnusson, E. (2003). *JastAdd - an aspect-oriented compiler construction system*. Science of Computer Programming, 47(1):37-58.

Henderson A., & Harris, J. (2011). *Conversational Alignment.* Interactions Magazine, 18(3):75-79, ACM.

Imtiaz, N., Rahman, A., Farhana, E., & Williams, L. (2013). *Challenges with Responding to Static Analysis Tool Alerts.* In proceedings of the 16th International Conference on Mining Software Repositories (MSR), IEEE/ACM.

Johnson, B., Song, Y., Murphy-Hill E., & Bowdidge R. (2013). *Why don't software developers use static analysis tools to find bugs?* In proceedings of the 35th International Conference on Software Engineering (ICSE), pp. 672-681, IEEE.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingier, J., & Irwin, J. (1997). *Aspect-oriented programming.* In proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 220-242, Springer.

Knuth, D. (1968). *Semantics of context-free languages*. Mathematical Systems Theory, 2(2):127-145.

Lohmeier, S. (2016). *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).

Magnusson, E., Ekman, T., & Hedin, G. (2009). *Demand-driven evaluation of collection attributes*. Automated Software Engineering, 16(2):291-322.

Mărășoiu, M., Church, L., & Blackwell, A. (2015). *An Empirical Investigation of Code Completion Usage by Professional Software.* 26th Annual Workshop of the Psychology of Programming Interest Group (PPIG).

Mori, M., MacDorman, K. F., & Kageki, N. (2012). *The Uncanny Valley [From the Field].* IEEE Robotics & Automation Magazine, 19(2):98-100.

Nachtigall, M., Nguyen Quang Do, L., & Bodden, E. (2019). *Explaining Static Analysis - A Perspective.* In proceedings of the 34th International Conference on Automated Software Engineering Workshop (ASEW), IEEE/ACM.

Novick, D. G., Hansen, B., & Ward, K. (1996). *Coordinating Turn-taking with Gaze.* In proceedings of the 4th International Conference on Spoken Language Processing (ICSLP), pp. 1888-1891.

Oulasvirta, A., Kristensson P. O., Bi X., & Howes A. (2018). *Computational Interaction*. Oxford University Press.

Pask, G. (1976). *Conversation Theory: Applications in Education and Epistemology*. Amsterdam and New York, Elsevier.

Rogers, Carl R., and Richard Evans Farson. 2015. *Active Listening*. Martino Publishing.

Sacks, H., Schegloff, E. A., & Jefferson, G. (1978). *A Simplest Systematics for the Organization of Turn-taking for Conversation*. Studies in the Organization of Conversational Interaction, pp. 7-55, Academic Press.

Sadowski, C., Söderberg, E., Church, L., Sipko, M., & Bacchelli, A. (2018). *Modern Code Review: A Case Study at Google.* In proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 181-190, ACM.

Searle, John. 1996. *The Construction of Social Reality*. Reprint edition. Penguin.

Söderberg, E., & Hedin, G. (2011). *Automated Selective Caching for Reference Attribute Grammars*. In proceedings of the International Conference on Software Language Engineering, pp 2-21, Springer Berlin Heidelberg.

Tanimoto, S. (1990). *VIVA: A Visual Language for Image Processing.* Journal of Visual Languages and Computing, 1(2):127-139, Elsevier.

Öqvist, J. (2018). *ExtendJ: Extensible Java Compiler*. In the conference companion of the 2nd International Conference on Art, Science, and Engineering of Programming, pp. 234–235, ACM.

Appendix A: Probe Questions

To help elaborate the conversational properties of an interaction we used the following questions, following the strategy originally adopted by the Cognitive Dimensions questionnaire (A. F. Blackwell and Green 2000). We list them here not as a canonical set, but to support discussion about what other questions would be helpful to ask.

Context
Who is having the conversation?
Where are they having it?
Why are they having it?
What language are they speaking?

Turns and temporality
How is speaking sequenced?
Do speakers take turns, or say things that come into their head?
How do the speakers know when it's their turn to talk?
How do speakers signal to others that now would be a good time for them to join in?
Can speakers interrupt each other? If speakers end up talking on top of each other, what happens?

How is the pace controlled?
How do speakers and listeners negotiate the speed of the conversation? What signals are there for speed up / slow down?
Who waits for whom in the conversations?

Meaning-making
Which parties of the conversation understand what aspects of the conversation?
How do they come to have that understanding?
Which words do they share a common understanding of?

Tolerance, breakdown & repair
How precise does the description of the elements need to be for the conversation to continue?
How much ambiguity can be absorbed and the conversation continue?
How do breakdowns and divergences in the understanding of the participants become apparent?
When these breakdowns do occur, what happens next?
What happens when conversations can't be repaired?

Explicability
Who in the conversation can ask for more information? How do they do it?
Who in the conversation can provide more information? What kind of information can they provide?

Side-channels & deixis
What other channels of information transfer exist outside the words being said?
How much are people aware of their communication in these channels? How much can they control them (e.g. intentional actions such as nodding, vs physiological ones such as pupil dilation)
Is this information used primarily for substance or for meta-properties of the conversation?
Is there any way outside the conversation of directing the focus of attention within the conversation?