An empirical investigation of code completion usage by professional software developers

Mariana Mărășoiu — *Computer Laboratory, Cambridge University, mcm79@cam.ac.uk*

Luke Church — *Google, luke@church.name*

Alan Blackwell — *Computer Laboratory, Cambridge University, Alan.Blackwell@cl.cam.ac.uk*

Abstract

Code completion is a widely used feature of modern integrated development environments. This study examines the ways in which code completion is used by professional software developers, as well as their actions when code completion doesn't offer the expected results. We observe that code completion is used with the intention of writing code faster and more correctly, that a large fraction of the code completions are not accepted by the users and that users often used code completion as a debugging tool: when the suggestions are not useful or not expected they are seen as a indicator that there is an error in the program.

Keywords: POP-II.A. Professional programmer, POP-II.B. Coding, POP-III.B. Dart, POP-III.D. Editors, POP-V.B. Protocol analysis

1. Introduction

Autocomplete for source code, also known as code completion, is an important part of modern integrated development environments (IDEs). Aimed at helping programmers write code faster, a code completion system offers a menu with a list of possible suggestions, usually alphabetically ordered. From the given list, the user can either choose one of the options, or they can ignore the list and resume typing. The completion list appears automatically when the user types a trigger character (such as '.' or '(' or '::', depending on the programming language and IDE), or it can be invoked with a specific keyboard binding. Intellisense[1] from Microsoft Visual Studio and Content assist[2] from Eclipse are two code completion system examples found in their respective IDEs.

A study of the Eclipse IDE observed that programmers used code completion as often as Copy & Paste (Murphy et al. 2006). This finding supports the general perception that code completion is a feature heavily used by programmers. We suggest that studying how professional software developers use code completion in their coding practices is a good starting point for future research on improvements of code completion systems and interfaces.

There are two common strategies for populating the list of possible completions: lexical and 'semantic'. Lexical models of completion look for other code starting with the same prefix as code the user is completing. Semantic models use a model of the grammar of the language to constrain the list of suggestions to those that are legal at the point of edit, e.g. by filtering out private methods. In this project we have studied a tool that provides semantic completions.

We studied the code completion usage patterns of 6 professional software developers, with each session organized in two parts. In the first part, the participants worked on a programming task for 30 minutes. The second part consisted of a retrospective think aloud session with each participant, where the recording of the first part was played back and the participant was asked to discuss what they were thinking or trying to do for each interaction with the code completion window. We report on the patterns of interaction we observed from analysing this data.

[1] https://msdn.microsoft.com/en-us/library/hcw1s69b.aspx

[2] http://help.eclipse.org/luna/topic/org.eclipse.platform.doc.isv/guide/editors_contentassist.htm

Our objective was to explore the following research questions:

1. What are the intentions of programmers when using code completion?
2. What are the behaviours that programmers engage in with respect to code completion?
3. What are the ways in which the interaction with the code completion popup can fail?
4. What recovery actions do programmers take when faced with such disruptions?

2. Related work

Murphy et al.'s (2006) study on how Java programmers are using the Eclipse IDE reports that all their participants used code completion and that its usage is at a similar level to Copy & Paste. However, they give no detail as to how and why code completion is used. In recognition of the importance of code completion within a programmer's environment, there has been a fair amount of research into trying to make autocomplete systems faster and more "intelligent". The informal motivation for such research is to reduce the cognitive load of the programmer who has to search for the appropriate completion of the method call or variable name. These systems focus on moving the most relevant suggestions at the top of the menu (i.e. terms that have the highest probability of being the ones that the programmer wants to use), thus reducing the need for actively searching or filtering the list.

Robbes and Lanza (2008) discuss ways in which the program history can help in providing the user with more relevant completions. Bruch et al. (2009) use data mining of multiple code repositories to provide more relevant method call suggestions. Similarly, Calcite (Mooty et al. 2010) offers suggestions for class instantiations extracted from other existing code, but it also allows for user edits of the completion list. Another type of enhancement is offered by Seahawk (Bacchelli et al. 2012) and Blueprint (Brandt et al. 2010), who address the problem of finding source code snippets that answer a user's query within the development environment.

The systems above focused on improving code completion with the goal of enhancing programmer performance (i.e. increased speed when typing source code). However, the purpose of using code completion seems to be more complex than writing faster code. Church et al. (2010) observe that code completion can be seen as a source of real-time feedback, both when it offers suggestions and when it is failing to do so. They suggest that the programmers are interpreting the failures of the code completion system as a sign that there are errors in their code.

Another perspective is to consider code completion as a form of negotiated interruption. As defined by *the taxonomy of interruptions* (McFarlane 1997, McFarlane & Latorella 2002), a negotiated interruption allows the user to decide when to deal with the interruption. The case of code completion has some resemblance to negotiated interruption systems, because the user can choose whether or not to interact with it (simply continuing to type), and if they do, how and when (returning to the code completion list by retriggering it).

Other previous literature suggests that programmers use code completion for the exploration of APIs (Stylos & Clarke 2007) and for faster code writing (Bruch et al. 2009). Further, Ward et al. (2012) mention spelling and locating known items as two main themes in the use of autocomplete for a library website. We use these previous studies as the starting point for developing the categories for the intentions of programmers when interacting with the code completion window.

3. Experimental design

The criteria we used for selecting a programming environment were: an existing user population with experience in using the environment, support for code completion, and suitability of the environment for instrumentation. We selected Dart Editor (Dart Editor | Dart, n.d.), an IDE for the Dart programming language built on the Eclipse RCP platform. Dart Editor exposes an API that can be used to report events within the system relevant for UX experiments (Instrumentation | Dart, n.d.).

3.1 Participants

We studied 6 professional software developers. Two of them had no previous experience with Dart, whilst the other four write Dart as part of their work. Of the 4 professional software developers experienced with Dart, three of them had no previous experience with the libraries used in the task. We report findings across the group as a whole, as the sample size is too small for any comparison between the levels of experience with the environment.

3.2 Data collection

We recorded the computer screen used by the participants throughout the study, and also recorded spoken audio. During their day-to-day development many Google engineers writing Dart use a custom, internal-only version of Dart Editor that implements the same instrumentation API, recording the data for later analysis. Whilst the publicly available versions of the Dart Editor do not use this API, we built an additional plugin for this study that implemented it and logged to disk the events such as accepting or closing the completion popup.

This gives us the opportunity to compare the data that has been recorded in this controlled study to a larger corpus of instrumentation data collected in a typical professional context.

3.3 Procedure

There were two phases to the study, each of approximately 30 minutes duration.

In the first phase, the participant was asked to perform a programming task. The task was selected to require use of several libraries, including at least one library that had to be used in depth. We expected that this would result in a set of interaction behaviours that represent a typical aspect of the day-to-day development work done by professional software developers.

The task was chosen so that it requires the participants to only write Dart code, without any need for working with HTML and CSS. This has several advantages. First, it offers us consistency when studying the code completion popup, as some of its features may be missing or be different for HTML and CSS. Second, because the participants do not need to switch contexts between files having different syntax, they will be able to focus more consistently on the aspects of the code that are supported by the semantic completion capabilities of Dart. We also avoid having the participants spend too much time on debugging activities and web designing. Whilst this decision limits the study to a specific part of the development cycle, it maximises the amount of code completion data that can be gathered within the experiment duration.

The required task was to implement the "ls" command for listing directory entries, including several of the following command line flags typically supported by implementations of "ls":

- "-a" to include entries starting with "."
- "-l" to specify a long listing format
- "-R" to list subdirectories recursively
- "-S" to sort the results by file size
- "-h" to print file sizes in human readable format rather than block numbers

Participants were asked to work at their normal pace, and were told that they were not expected to finish the entire task in the given time. They were asked to work on the parts of the task in any order. The participants were not asked to think aloud, although all of them chose to verbalize some of their thoughts as they were coding. They were informed that the purpose of the study was to observe the interaction with the tooling, but not specifically told that it was concerned with the code completion system.

The second part of the study consisted of a retrospective think aloud (RTA) session. The experimenter played back the screen recording that had been made during the previous task. Each time that the code completion popup appeared in the recording, the experimenter asked the participant what they had been thinking and doing at that time.

The retrospective think aloud protocol was first described and used by Hewett and Scott (1987) as a post-search review and is presented under various names in the literature: retrospective testing (Nielsen, 1994), retrospective report (Capra, 2002), think after (Branch 2000), retrospective verbal protocol and retrospective verbalisation (Kuusela & Paul, 2000). In a typical application, the participants first perform the assigned tasks in silence, followed by a verbalization of their thoughts after the completion of the task, usually supported by a video recording (Nielsen, 1994). A number of studies (Bowers & Snyder 1990, Van Den Haak et al. 2003, Nielsen 2002) have compared the results of concurrent and retrospective think aloud protocols. The general consensus of these studies is that there are few differences between the two approaches in terms of quality and quantity of data produced, but that the nature of the data gathered is somewhat different. For example, Bowers and Snyder (1990) report that concurrent think aloud results in more procedural verbalizations, whereas participants doing retrospective think aloud give more explanations. Further, it has been shown that retrospective think aloud is a valid method for finding out what the participants are thinking when solving an experimental task (Guan et al. 2006).

We used retrospective think aloud in order to address several problems of concurrent think aloud. First, programming is known to be a cognitively intensive task, and we expected interaction with the code completion window to be particularly demanding, involving search, exploratory understanding, and strategy formation. Thus we decided against using a concurrent think aloud protocol as we wanted to avoid disrupting the participants if they fell silent during these critical moments of the tasks. Using retrospective think aloud we were able to gather information about each interaction we were interested in without risking this disruption. Second, we wanted to collect instrumentation data that was as similar as possible to how professional software developers would use code completion in the wild. Hence we only informed the participants of our interest in the use of code completion at the beginning of the retrospective think aloud phase, in order to avoid sensitizing them in a way that might change their behaviour whilst coding.

Nevertheless, one of the reported problems of retrospective think aloud is that the participants may not remember what they were thinking. It was in order to avoid this, we showed participants the video of the previous task and prompted them to talk for each individual code completion. Another observed problem is that the participants can give different explanations for what they were thinking or doing during the retrospective think aloud. In order to account for these, during the analysis, we consider both the concurrent and retrospective vocalizations of the participants, as well as their observed behaviour from tool instrumentation.

4. Analysis method

4.1 Protocol analysis

The 6 sessions resulted in 192 instances of interaction with the code completion window. The concurrent and retrospective verbalizations for each interaction were transcribed into text files. The user interface actions of the participants were also transcribed using the screen recording from the first part of the study, together with the instrumentation logs. These 3 transcripts, for each of the 192 interaction instances, provided the corpus of data for coding.

This data was coded according to 4 aspects, which created our coding frame, as can be seen in Table 1. Each aspect corresponds to one of the research questions described in Section 1. Since each coded aspect describes a different aspect of interaction with the code completion window, a single interaction could potentially be assigned codes relating to multiple different aspects. Further, we observed that the users sometimes reported multiple intentions motivating their use of code completion, in addition to multiple behaviours during their actual interaction. We therefore allowed the assignment of multiple codes from the same aspect to each coded interaction.

After preprocessing of the data and selecting the main categories of the coding frame, we used both deductive and inductive qualitative content analysis in order to develop the set of codes (Mayring, 2000). The former was used to identify User Intentions, as our seed codes were derived from reasons for autocomplete usage that had been mentioned in previous literature: Exploration (Stylos & Clarke

2007), Speed (Bruch et al. 2009), Correctness and Search (Ward et al. 2012). Because there were further themes in the data beyond those covered by these seed codes, we also used inductive analysis to generate new codes. For the remaining three aspects we used inductive qualitative content analysis, with the codes emerging from the raw data.

Table 1. Number of codes in the frame and assigned to the data per coded aspect

Coded aspect	# codes in the frame	# codes used in the data
User Intention	7	286
Interaction Behaviour	8	247
System Disruption	6	69
Recovery Action	7	63

To confirm the reliability of the resulting coding frame, an additional researcher was recruited in order to independently code the qualitative data (transcripts, video and screen recordings) based on the developed coding frame. The researcher coded 48 of the interaction instances, representing 25% of the data. The interrater reliability calculation returned an average of 95.14% for percent agreement and a value of 0.81 for Krippendorff's alpha. On Krippendorff's (1980) scale, agreement $\alpha \geq 0.8$ is considered as reliable.

4.2 Pattern discovery

In order to be able to easily recognize patterns and co-occurrences of the codes in the data, we transformed it into a visual form. This new representation allows for visualising the entirety of the data at once, where each interaction with the code completion window is visualised as a glyph, containing elements that correspond to all the codes assigned to that interaction. We organize our glyps in a tabular form, similar to the representation of charts in small multiples (Tufte, 1990) in order to allow for comparisons. Glyph representation has been also suggested by Gehlenborg and Brazma (2009) as a useful method for visualising and comparing between hundreds of samples.

Figure 1 presents an example of a glyph for a single coded interaction, where each colored square within the glyph represents one of the codes assigned to the interaction. The horizontal position and color of the square represent the aspect that is being coded, whilst the vertical position represents the index of that code within the coding scheme (see Tables 2-5). Figure 2 presents a subset of the data represented using the model from Figure 1. Using this visualization we were able to identify patterns and co-occurrences of codes by observing repeated instances of glyphs that were visually similar. These patterns are discussed in Section 5.

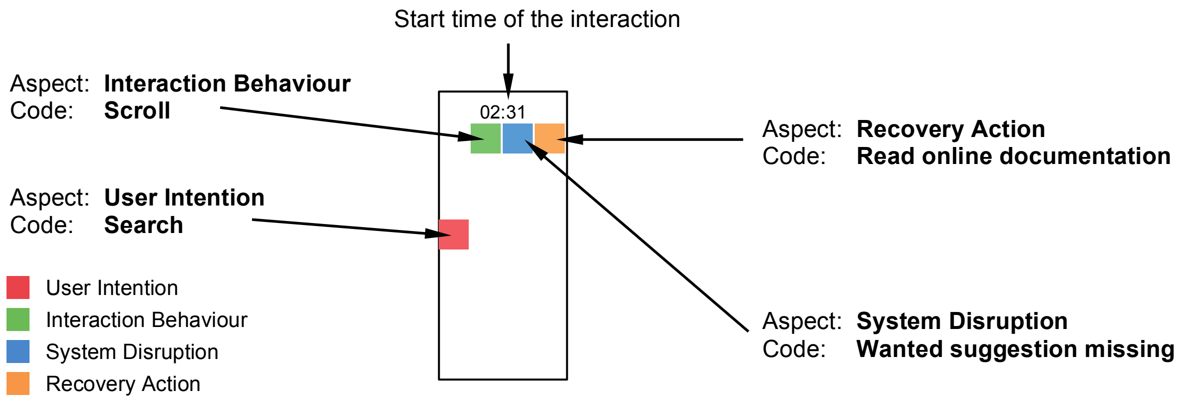
Start time of the interaction
Aspect: Interaction Behaviour
Code: Scroll
02:31
Aspect: Recovery Action
Code: Read online documentation
Aspect: User Intention
Code: Search
User Intention
Interaction Behaviour
System Disruption
Recovery Action
Aspect: System Disruption
Code: Wanted suggestion missing

Figure 1. A glyph summarising the codes assigned to a single instance of interaction with the code completion window. Each colored square encodes a code (row) of the given aspect (column).

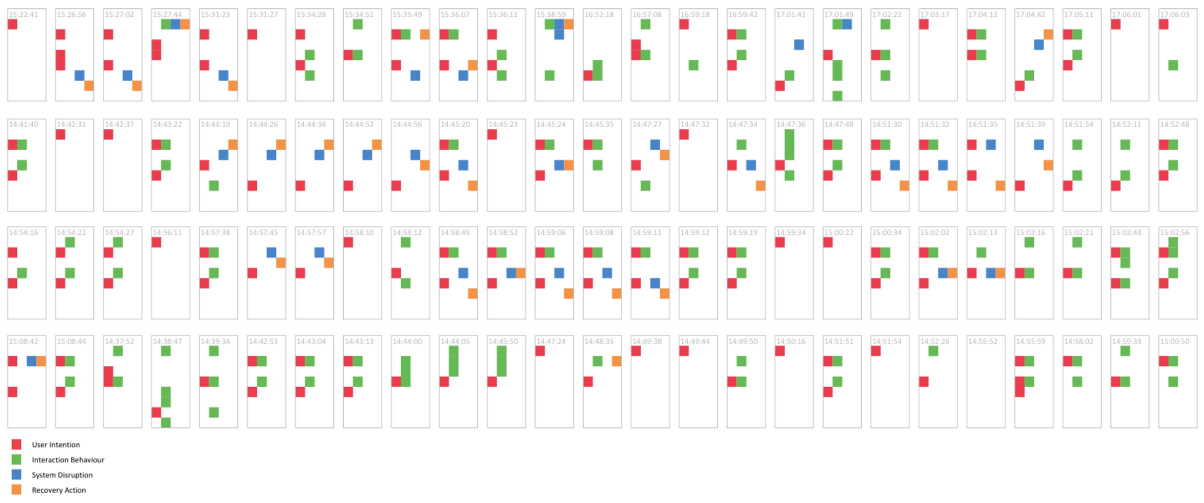

Figure 2. Example of analysis visualisation with multiple interaction coding glyphs.

In order to compute the association between codes that were part of the patterns we identified visually, we used the normalized pointwise mutual information (npmi) statistic (Bouma 2009). A score of -1 between two codes means that they never occur together, a score of 0 represents independence and +1 represents complete co-occurrence in the data. The npmi statistic is used to verify the validity of the patterns observed through analysing the visualisation.

5. Results

5.1 What are the intentions of programmers when using code completion?

Table 2 presents the complete description of the set of codes used for coding programmer intentions when interacting with the code completion window, i.e. the User Intention category. The descriptions for each code are based on what we observed in our data during the pre-coding phase.

We would like to note the distinction that we drew between exploration and search behaviours. The UI-Exploration code represents undirected exploration, i.e. when the user doesn't have a specific goal in mind. We considered directed exploration as a form of search, because the user was looking for something that would help them achieve a given goal. The merging between search and directed exploration in one code, UI-Search, is due to directed exploration behaviour and search behaviour being indistinguishable from each other in the data we gathered.

However, Stylos and Clarke mention that their participants used "code-completion as the primary means of exploration" (2007), more specifically, to gain confidence that the instantiated object was the correct one and to find out which "methods or properties perform the needed functions". Since their participants had a goal that they wanted to achieve, we consider that this description fits better the category of search or even directed exploration, rather than (undirected) exploration.

Interestingly, one of the main findings so far is the lack of API exploration in the participants' intentions. From the total 192 instances of interaction with the code completion window, only 3 of these were coded as UI-Exploration. We suggest that this is due to the participants being focused on working on the task and having a goal in mind that they tried to achieve through the use of code completion. As such, their interactions with unfamiliar APIs were mostly about searching for a method or attribute that would perform a desired action. However, during the Retrospective Think Aloud conversations, one of the participants mentioned the use of code completion for API exploration in his work:

> *"I definitely use autocomplete. To explore APIs. Because it's so... not a lot of times you have a good idea of what you want, especially if you're like, what getters do I have, right? it's gotta be helpful."* *ID3, Experienced with Dart*

For another participant, the distinction between search and exploration was less clear:

Here, you know, we have a list and I was trying to figure out what I can do with the list. I was trying to get one item, I guess, I don't remember exactly, but I was trying to see what I can do with the list. And it was showing me. At first I get the first one, yeah, and I said, yeah, probably I shouldn't use the first one, and after that that I said, maybe I should do a for loop.

ID4, Inexperienced with Dart

Table 2. Codes and their descriptions for the User Intention category.

User Intention	**# instances**	**Description**
UI-Ignore	29	The user doesn't interact with the code completion window, they dismiss it as soon as it appears, or they interact with it, but don't have any intention of using it. For example: • Idle interaction (the user interacts with the code completion window whilst thinking about what to do next). • The user does the sequence of keystrokes that they would have done in the code editor, but these were grabbed by the code completion window
UI-Correctness and/or Speed	90	The user wants to ensure that the name of the method is correct (the method exists, getting capitalisation right) The user has typed the first part of the attribute or method name and uses code completion to fill in the rest
UI-Exploration	3	The user isn't looking for something specific and doesn't have a goal in mind, they're just getting to know the API
UI-Search	68	The user knows what they want to achieve, but doesn't exactly know how to get there. For example: • The user doesn't know the exact name of the method, but knows possible alternatives • The user is looking for a specific property on an object, or a way to get it if it's not an attribute of an object
UI-Test	9	The user is triggering code completion to see whether the tooling is working properly
UI-Documentation	4	The user is triggering code completion to read documentation

During the first step of the coding phase, we observed that it was very difficult to differentiate between when participants were using code completion to achieve speed, correctness or both. As such, we decided to combine the two seed codes into a single one, UI-Correctness and/or Speed. Further, as can be seen in Table 2, correctness and speed was one of the main intentions when using the code completion window. This confirms findings in previous literature – progammers use code completion in order to write less error prone code, whilst, at the same time, making the writing process easier and faster.

"So like, if the IDE fills in a method name for me, then I know that I didn't make a typo for example. So I have this many bugs less in my code, I'm sure. Right, because it doesn't make mistakes. So what I usually want to type in every method with autocomplete. Or like, most of the times."

ID1, Experienced with Dart

We also observed that users use code completion to test whether their code is working, whether an import is behaving, or, in the case of software developers new to the language and the editor, to simply check that code completion exists. This helps them check the state of the environment and the tooling.

"I realized I need to go back to Directory, and then went 'dot' to see what's happening. Cause this feature [code completion], I love it, you know. It's something that I would like to see like, you know, in all the editors. And I was assuming that it should be here."

ID4, Inexperienced with Dart

5.2 What are the behaviours that programmers engage in with respect to code completion?

The behaviours that we have identified are presented in Table 3, as well as how many of the 192 interactions with the code completion window from the participants were labelled with each code.

We observe that a fairly typical way of interacting with the code completion is to filter the list of suggestions and then accept one of them, either with Enter or with a mouse click. In fact, 28.1% of all the interactions with the code completion window consisted of filtering and accepting a suggestion. The strength of co-occurrence between the IB-Filter and IB-Accept codes, as reported by the npmi statistic, is 0.39, which suggests a correlation between them.

Table 3. The codes for the Interaction Behaviour category, the number of instances coded and a description for each code.

Interaction behaviour	# instances	Description
IB-Scroll	35	The user goes through the list of code completions using mouse/trackpad scroll or arrow down/up.
IB-Filter	81	The user filters the code completion list by typing
IB-Unfilter	8	The user deletes a character at the end of the method name whilst code completion window is displayed, thus expanding the list of completions shown
IB-Accept	78	The user accepts a suggestion from the completion list
IB-Dismiss	20	The user actively dismisses the code completion window (with ESC or clicking somewhere else in the code editor)
IB-ManualTrigger	21	The user triggers code completion manually (either by deleting an existing '.' or by using the keyboard shortcut)
IB-ReadDocumentation	3	The user reads the documentation for an attribute from the documentation popup of the code completion window
IB-InspectParameters	4	The user looks at the parameters or return types of a method to see whether the method is useful or if the parameters need default-value overriding

Further, the filter-accept behaviour also has a good correlation with the intention of using code completion for speed and correction (npmi score of 0.50). For example, one participant mentioned in the second part of the study, as they were watching their filter-and-accept behaviour:

"startsWith.. I know what I'm looking for here. It helps me get capitalization correct as well, is the other, another primary reason for dealing with it."

ID5, Experienced with Dart

Scrolling through the list of suggestions was another behaviour that the participants often engaged in, and it has a good association with the intention of searching for something (npmi score of 0.39).

5.3 What are the ways in which the interaction with the code completion popup can fail?

We have identified a list of disruptions that can occur whilst using code completion, which are described in Table 4.

From the preliminary results, only 40.1% of all triggered code completion windows resulted in accepting a suggestion. By observing what the user was doing when they decided not to accept a completion we see that is partly due to the users not being familiar with some of the APIs used, and thus they engage more in searching for an item which then they don't find in the code completion window. This goes some way to explain why SD-MissingSuggestion code accounts for 19% of all the disruptions recorded.

However, in a sample of 10,000 completions collected from instrumented usage of Dart within Google, this fraction only rises to 44.2%. This suggests that finding items may remain a common challenge even for developers experienced with the APIs.

Table 4. The codes for the System Disruption category, the number of instances coded and a description for each code.

System disruption	**# instances**	**Description**
SD-MissingSuggestion	14	The wanted suggestion is missing
SD-NotExpected	15	The list of suggestions is unexpected or surprising to the user
SD-NoAppear	15	Completion window fails to appear
SD-Disappeared	17	Completion window disappeared undesired
SD-AccidentalDismiss	2	The user accidentally dismisses the code completion window
SD-FailedAccept	5	The user wanted to accept a completion but accidentally dismissed the code completion window

An important part of system disruptions is due to the user not finding the expected suggestions in the list (SD-NotExpected) and to the failures of the code completion window (SD-NoAppear and SD-Disappeared). We mentioned previously that code completion can be viewed as a negotiated interruption in the programmer's workflow, albeit a very useful and often desired interruption. We argue that, in situations where the code completion engine fails to provide relevant suggestions, this interruption aspect is more evident: the failure is a signal that the code may not be correct, and the user can choose whether to try to find out what the problem is, or to continue typing without the aid of code completion. It should also be noted that this aspect of code completion is often overlooked when designing systems or improvements for code completion systems.

> *"Oh, and 'fse'*[3] *doesn't work here, which is interesting. I think it should. But the fact that it doesn't makes me actively go back. I would rather go back and fix the inference problem than carry on. Even though I'm right, just to increase the confidence value."*
>
> *ID5, Experienced with Dart*

5.4 What recovery actions do programmers take when faced with such disruptions?

The possible recovery actions that we have identified during our studies are presented in Table 5. The recovery actions that the programmers engage in correspond to the way in which the system disrupted their coding activity. For example, if the suggestion they were looking for is missing, then the programmer goes to read the online documentation (npmi score of 0.53) or tries to change the code that they're writing (npmi score of 0.33), as the absence of the wanted suggestion signals them that the object they have is not useful.

[3] [n.a.] An instance of FileSystemEntity, a type of object for manipulating files.

“’flags’ doesn't seem to exist, so what do I get? Command, hashCode, name, so.. I'm actually not sure. So I'm gonna look at addFlag.” [goes to the online documentation]

ID3, Experienced with Dart

Table 5. The codes for the Recovery Actions category, the number of instances coded and a description for each code.

Recovery action	**# instances**	**Description**
RA-Documentation	7	User goes to look for documentation
RA-ChangeCode	26	User decides that the code they were trying to write wouldn’t work and tries to write different code
RA-WrongProgram	6	User decides that some other code in their program is wrong and tries to fix it
RA-Override	8	User decides that code completion is misleading and continues regardless
RA-ResummonForAccept	1	Resummon completion to accept it
RA-ResummonForObserve	13	Resummon completion to look at what needs to be different
RA-ManualComplete	3	The user continues typing the name of the attribute, after accidentally dismissing the code completion window

We have also observed that if the list of suggestions is surprising or unexpected to the user, they use it as a signal that either the code they’re writing is wrong (npmi score of 0.31) or that some other code in their program is wrong (npmi score of 0.73).

“Again, here, I use code completion and it tells me that nothing works. And that tells me.. When I look at it it's because I'm missing a close brace so there it is helping. The fact that it doesn't work is telling me something's wrong with the rest of my code.”

ID5, Experienced with Dart

For example, due to Dart’s optional typing, programmers can choose whether to add or not to add type annotations to their code, with the program having the same semantics either way. An interesting behaviour that we observed was that the participants actively added types in order to get better suggestions. When the list of completions is surprising to the user (e.g. when it shows a restricted list of attributes), then the programmer realizes that type inference failed and then they modify their code by adding types to some of their variables in order to trigger more relevant suggestions.

“The problem with this was that I didn't put in the type. So I was like, that's odd.”

ID2, Experienced with Dart

These behaviours of using the absence of information in the completion dialog as a prompt to change their program are made possible by the semantic (as opposed to lexical) completion model.

As expected, we observed that when a user accidentally fails to accept a suggestion, they engage in two main possible types of behaviour: they continue typing without needing help from a code completion window (npmi score of 0.87), or they manually trigger again code completion in order to accept the desired option from the list (npmi score of 0.69).

Similarly, when the code completion window disappears as the programmer was filtering or interacting with it, they will either continue regardless of the state of code completion (npmi score of 0.53) or they will try to re-trigger it again in order to observe what went wrong and what needs to be different (npmi score of 0.69).

6. Conclusions

Whilst limited in scope and sample size, this study provides some empirical evidence of how professional software developers use code completion in their work. We coded and described four aspects of their interaction with the code completion: intentions, behaviours, breakdowns and recovery actions. We have identified common co-occurrences of codes.

During the study, participants used code completion extensively even though they weren't prompted to do so. This is confirms the findings of Murphy et al. (2006), which suggest heavy usage of code completion in programming work. However, we didn't observe code completion being used as an exploration tool as frequently as has been reported in previous work (e.g. Stylos and Clarke, 2007).

One of the principal patterns that we found was the correlation between participants using code completion with the intention of writing code faster and more correctly. Further, their typical behaviour when having this intention was to filter the suggestion list and accept one of the few remaining options.

We also observed the main disruptions that can occur when interacting with code completion, as well as a set of recovery actions that participants engage in after such breakdowns. For example, not finding the desired completion had good correlation with the participants reading online documentation afterwards. We observed that many of the completions offered were not accepted by the participants, and that this behaviour is consistent with professional usage. This suggests that there is considerable room for improvement in the use of code completion as a search tool.

We noted the use of code completion failures as a static debugging tool confirming previous work (Church et al., 2010). When code completion didn't offer any helpful suggestions, the participants considered this an indicator of an error elsewhere in the program, and switched to fixing their code.

We suggest that our findings might be used as a starting point for improving code completion systems. Given the observed high prevalence of code completion use as a debugging aid, further work might investigate how code completion systems can better support this usage.

7. References

Bacchelli, A., Ponzanelli, L., & Lanza, M. (2012, June). Harnessing stack overflow for the IDE. In Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on (pp. 26-30). IEEE.

Bowers, V. A., & Snyder, H. L. (1990, October). Concurrent versus retrospective verbal protocol for comparing window usability. In Proceedings of the Human Factors and Ergonomics Society Annual Meeting (Vol. 34, No. 17, pp. 1270-1274). SAGE Publications.

Branch, J. L. (2000). Investigating the information-seeking processes of adolescents: The value of using think alouds and think afters. Library & Information Science Research, 22(4), 371-392.

Brandt, J., Dontcheva, M., Weskamp, M., & Klemmer, S. R. (2010, April). Example-centric programming: integrating web search into the development environment. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 513-522). ACM.

Bruch, M., Monperrus, M., & Mezini, M. (2009, August). Learning from examples to improve code completion systems. In Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (pp. 213-222). ACM.

Bouma, G. (2009). Normalized (pointwise) mutual information in collocation extraction. Proceedings of GSCL, 31-40.

Church, L., Nash, C., & Blackwell, A. F. (2010). Liveness in notation use: From music to programming. Proceedings of PPIG 2010, 2-11.

Capra, M. G. (2002, September). Contemporaneous versus retrospective user-reported critical incidents in usability evaluation. In Proceedings of the Human Factors and Ergonomics Society Annual Meeting (Vol. 46, No. 24, pp. 1973-1977). SAGE Publications.

Dart Editor | Dart. (n.d.). Retrieved June 14, 2015, from www.dartlang.org/tools/editor/

Gehlenborg, N., & Brazma, A. (2009). Visualization of large microarray experiments with space maps. BMC Bioinformatics, 10(Suppl 13), O7.

Guan, Z., Lee, S., Cuddihy, E., & Ramey, J. (2006, April). The validity of the stimulated retrospective think-aloud method as measured by eye tracking. In Proceedings of the SIGCHI conference on Human Factors in computing systems (pp. 1253-1262). ACM.

Hewett, T. T., & Scott, S. (1987, November). The use of thinking-out-loud and protocol analysis in development of a process model of interactive database searching. In Human Computer Interaction–INTERACT (Vol. 87, pp. 51-56).

Instrumentation | Dart. (n.d.). Retrieved June 14, 2015, from https://github.com/dart-lang/bleeding_edge/tree/80988a506bdf74d2e87bf0d7e1211d035c2e5624/dart/editor/tools/plugins/com.google.dart.engine/src/com/google/dart/engine/utilities/instrumentation

Krippendorff, K. (1980). Content Analysis: An Introduction to its Methodology. Sage Publications

Kuusela, H., & Paul, P. (2000). A comparison of concurrent and retrospective verbal protocol analysis. The American journal of psychology, (113), 387-404.

Mayring, P. (2000). Qualitative Content Analysis [28 paragraphs]. Forum Qualitative Sozialforschung / Forum: Qualitative Social Research, 1(2), Art. 20

McFarlane, D. C. (1997). Interruption of people in human-computer interaction: A general unifying definition of human interruption and taxonomy (No. NRL-FR-5510-97-9870). Office of naval research arlington va.

McFarlane, D. C., & Latorella, K. A. (2002). The scope and importance of human interruption in human-computer interaction design. Human-Computer Interaction, 17(1), 1-61.

Mooty, M., Faulring, A., Stylos, J., & Myers, B. A. (2010, September). Calcite: Completing code completion for constructors using crowds. In Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on (pp. 15-22). IEEE.

Murphy, G. C., Kersten, M., & Findlater, L. (2006). How are Java software developers using the Eclipse IDE?. Software, IEEE, 23(4), 76-83.

Nielsen, J. (1994). Usability engineering. Elsevier.

Nielsen, J., Clemmensen, T., & Yssing, C. (2002, October). Getting access to what goes on in people's heads?: reflections on the think-aloud technique. In Proceedings of the second Nordic conference on Human-computer interaction (pp. 101-110). ACM.

Robbes, R., & Lanza, M. (2008, September). How program history can improve code completion. In Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on (pp. 317-326). IEEE.

Stylos, J., & Clarke, S. (2007, May). Usability implications of requiring parameters in objects' constructors. In Proceedings of the 29th international conference on Software Engineering (pp. 529-539). IEEE Computer Society.

Tufte, E. (1990). Envisioning Information.

Van Den Haak, M., De Jong, M., & Jan Schellens, P. (2003). Retrospective vs. concurrent think-aloud protocols: testing the usability of an online library catalogue. Behaviour & Information Technology, 22(5), 339-351.

Ward, D., Hahn, J., & Feist, K. (2012). Autocomplete as research tool: A study on providing search suggestions. Information Technology and Libraries, 31(4), 6-19.