Managing Software Change Tasks: An Exploratory Study

Jonathan Sillito, Kris De Volder, Brian Fisher, Gail Murphy
Department of Computer Science
University of British Columbia
{sillito, kdvolder, fisher, murphy}@cs.ubc.ca

Abstract

Programmers often have to perform change tasks that involve unfamiliar portions of a software system's code base. To help inform the design of software development tools intended to support programmers in this context, we conducted a qualitative study of how programmers manage such change tasks. In the study we observed Java programmers using a state-of-the-practice IDE to work on real change tasks to a medium-sized open source software system. In this paper we present our results, describing eight observations about the programmers' behavior and the impact of the development environment on their behavior. We also highlight several key challenges faced by the programmers and discuss the implications of our results on the design of development tools.

1 Introduction

Programmers often have to perform change tasks that involve unfamiliar portions of a software system's code base. This situation arises when newcomers join a development team and are assigned change tasks to learn about the system and the project [11]. This situation also arises for experienced programmers when making changes having a non-localized impact, such as might occur when changing an API used by other programmers.

Aspects of today's development environments are aimed at supporting programmers in these situations as they learn about and work with a code base. However, this support has its limitations and programmers working on change tasks continue to experience significant difficulties [6]. To explore this situation, we undertook a qualitative grounded theory study in which we observed experienced Java programmers use a state-of-the-practice Java development environment to work on real change tasks to a medium-sized open source software system. Our analysis focused on the goals of the programmers in performing these tasks and on how the tools provided by the development environment were used to accomplish these goals.

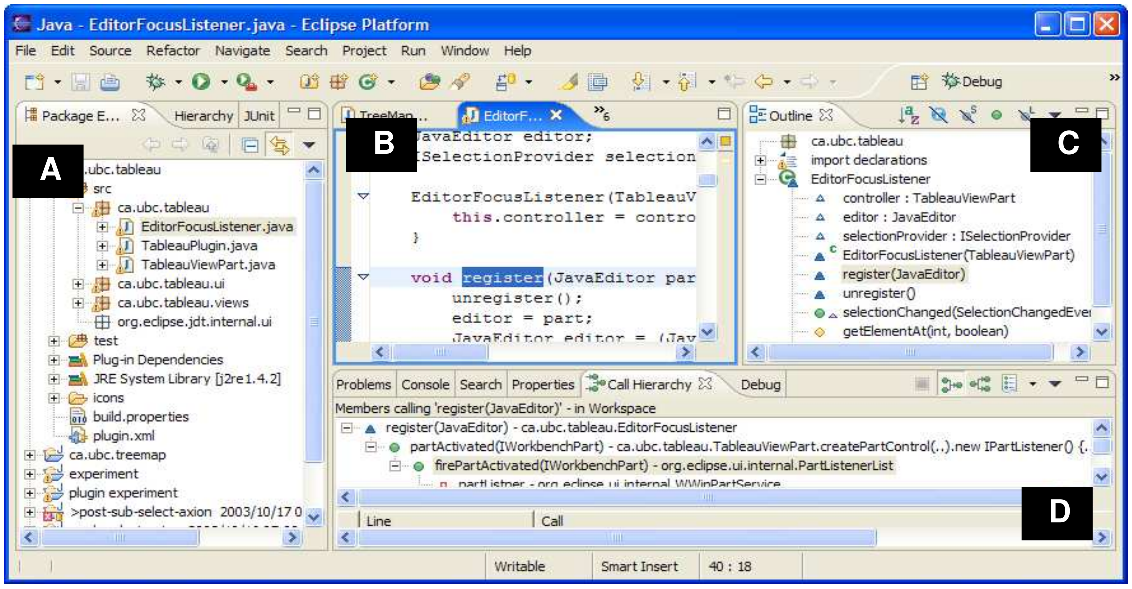
In this paper, we present three contributions resulting from this study. First, we report eight observations that characterize the activities the programmers performed to manage change tasks and how they were influenced by the development environment. Example observations include how programmers decomposed goals into ones that were supported directly by the tools and how programmers tried to minimize the amount of source code they needed to understand in detail. Second, based on these observations, we describe five key challenges that the programmers faced while performing the tasks, such as cognitive overload and working from false assumptions. Third, we describe four directions for tool development that may help overcome the challenges we have identified.

We begin with a comparison of our study to earlier work (Section 2). We then describe our study approach (Section 3) and the results (Section 4). Next we highlight the key challenges (Section 5.1) and our suggestions for development environment tools (Section 5.2). Finally we present the limitations of our study (Section 6) and end with a summary (Section 7).

2 Related Work

The situation of programmers performing change tasks has been studied from a number of perspectives. Some researchers have focused on proposing and validating cognitive models of how a programmer comprehends a program (e.g., [17]). Other work has attempted to bridge the gap between comprehension theories and tool design (e.g., [13, 18]). In comparison, we report on the broader process of how change tasks are managed, of which comprehension activities are just one piece. As a result, we are able to consider tool implications that go beyond aiding comprehension.

0-7803-9508-5/05/$20.00 ©2005 IEEE


Java - EditorFocusListener.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Package E…
Hierarchy
JUnit
EditorF…
Outline
A
B
C
D
Problems Console Search Properties Call Hierarchy Debug
Line
Call
Writable
Smart Insert
40 : 18

```
ubc.tableau
  src
    ca.ubc.tableau
      EditorFocusListener.java
      TableauPlugin.java
      TableauViewPart.java
    ca.ubc.tableau.ui
    ca.ubc.tableau.views
    org.eclipse.jdt.internal.ui
  test
  Plug-in Dependencies
  JRE System Library [j2re1.4.2]
  icons
  build.properties
  plugin.xml
ca.ubc.treemap
experiment
plugin experiment
>post-sub-select-axion 2003/10/17 0
```

```
JavaEditor editor;
ISelectionProvider selection

EditorFocusListener(TableauV
    this.controller = contro
}

void register(JavaEditor par
    unregister();
    editor = part;
    JavaEditor editor = (Jav
```

```
ca.ubc.tableau
import declarations
EditorFocusListener
    controller : TableauViewPart
    editor : JavaEditor
    selectionProvider : ISelectionProvider
    EditorFocusListener(TableauViewPart)
    register(JavaEditor)
    unregister()
    selectionChanged(SelectionChangedEve
    getElementAt(int, boolean)
```

```
Members calling 'register(JavaEditor)' - in Workspace
register(JavaEditor) - ca.ubc.tableau.EditorFocusListener
  partActivated(IWorkbenchPart) - ca.ubc.tableau.TableauViewPart.createPartControl(..).new IPartListener() {.
    firePartActivated(IWorkbenchPart) - org.eclipse.ui.internal.PartListenerList
```

Figure 1. A screenshot of the Eclipse Development Environment, with several major tools labeled: (A) package explorer, (B) source code editor with tabs, (C) content outline view, and (D) call hierarchy browser.

More similar to our study are efforts that qualitatively examine the work practices of programmers. For example, Flor et al. used distributed cognition to study a single pair of programmers performing a straightforward change task [3]. We extend their methods to a larger participant pool and a more involved set of change tasks with the goal of more broadly understanding the challenges programmers encounter. As another example, Singer et al. studied the daily activities of software engineers [12]. We focus more closely on the activities directly involved in performing a change task, producing a complementary study at a finer scale of analysis.

Three recent studies have focused more directly on the use of current development environments. Robillard et al. characterize how programmers who are successful at maintenance tasks typically navigate a code base [9]. Deline et al. report on a formative observational study also focusing on navigation [2]. Our study differs from these in considering the change management process rather than focusing exclusively on navigation. Ko et al. report on a study in which expert Java programmers used the Eclipse development environment to work on five maintenance tasks on a small program [6]. Their intent was to gather design requirements for a maintenance development environment. Our study differs in focusing on a more realistic situation involving a larger code base, more involved tasks and a study format that enables tracking of the reasons behind the actions and intent of the programmers, resulting in a different set of observations.

3 Study Method

The goal of our study was to characterize how programmers manage complex change tasks, including the way they use tools and the challenges they face. In the study, pairs of programmers performed assigned change tasks on a moderately-sized open-source system. We choose to study pairs of programmers because we believed that the discussion between the pair as they worked on the change task would enable us to find out *why* particular actions were being taken during the task, similar to earlier efforts (e.g., [3] and [7]). We studied how multiple pairs approached the same change task, and we studied different pairings of our participants.

To structure our data collection and analysis in this exploratory study, we used a grounded theory approach which has been described as an *emergent process* intended to support the production of a theory that "fits" or "works" to explain a situation of interest [4, 14]. In this approach, data collection, coding and analysis do not happen strictly sequentially, but are overlapping activities. As data is reviewed and compared, important themes or ideas emerge (called *categories*) that help contribute to an understanding of the situation. As cat-

Table 1. Study tasks. Numbers refer to IDs in the ArgoUML issue tracking system.

Task	Description
484	Make the font size for the interface configurable from the GUI.
1021	Add drag and drop support for changing association ends.
1622	Add property panel support for change, time and signal event types.
1622a	Add textual annotation support for change, time and signal event types.
2718	Fix a model saving error that occurs after a use case with extends relationships is deleted from the model.

egories emerge, further selective sampling and adaptive coding can be performed to gather more information. Further analysis aims to organize and understand the relationships between the identified categories, possibly producing higher-level categories in the process.

3.1 Study Setup

Participants in our study used the Eclipse Java development environment[1] (version 3.0.1) which is a widely used IDE. A screenshot of Eclipse is shown in Figure 1 showing several commonly-used views or tools: the package explorer (showing the package, file and class structure), the tabbed source code editor, the content outline view (showing the structure of the currently open file) and the call hierarchy browser. Other commonly-used views not shown in the screenshot include a type hierarchy view, a search results view, a breakpoint view, a variable view, and a launch view (which shows the execution stack while executing an application in debug mode).

The study involved twelve sessions (S1...S12). In each session two participants performed an assigned task as a pair working side-by-side at one computer. Following the terminology of Williams et al. [19], we use the term “driver” for the participant assigned to control the mouse and keyboard and “observer” for the participant working with the driver. In most sessions, the least experienced programmer was asked to be the driver.

In each session, the programming pair was given forty-five minutes to work on a change task. Participants were stopped after the forty-five minutes elapsed regardless of how much progress had been made on the task. An audio recording was made of discussion between the pair of participants, a video of the screen was captured, and a log was made automatically of the events in Eclipse related to navigation and selection. The experimenter, who was present during each session, then briefly interviewed the participants about their experience. The interviews were informal and focused on the challenges faced by the pair, their strategy, how they felt about their progress and what they would expect to do if they were continuing with the task. An audio recording of the interview was made.

[1]`http://www.eclipse.org`

3.2 Change Tasks

Table 1 describes the change tasks, which were all enhancements or bug fixes to the ArgoUML[2] code base (versions 0.9, 0.13 and 0.16). ArgoUML is an open-source UML modeling tool implemented in Java that comprises roughly 60KLOC. The tasks were completed tasks chosen from ArgoUML’s issue-tracking system. The tasks were based on complex, non-local changes. We did not expect that the participants would be able to complete the tasks in the time allotted, but we believed they would be able to make significant progress. Participants were asked to accomplish as much as possible on the one task, but not to be concerned if they could not complete the task. In four of the sessions, S4, S7, S11 and S12, participants were asked to work on the same task they had worked on in a previous session, allowing us to gather data about later stages of work on the task. Table 2 shows which tasks were assigned for each session.

3.3 Participants

Nine programmers (P1...P9) participated in our study. All participants were computer science graduate students with varying amounts of previous development experience, including experience with the Java programming language. Participants P1, P2 and P3 had five or more years of professional development experience. Participants P4, P5 and P6 had two or more years of professional development experience. Participants P7, P8 and P9 had no professional development experience, but did have one or more years of programming experience in the context of academic research projects.

[2]`http://argouml.tigris.org`

Table 2. Session number (SN), driver, observer and assigned task for each session.

SN	Driver	Obs.	Task	SN	Driver	Obs.	Task	SN	Driver	Obs.	Task
S1	P7	P3	484	S5	P5	P3	1622	S9	P8	P6	2718
S2	P4	P1	1622	S6	P5	P2	1021	S10	P8	P2	1622a
S3	P4	P7	1021	S7	P3	P1	1622	S11	P9	P2	1622a
S4	P6	P4	1622	S8	P7	P5	2718	S12	P9	P6	1622a

All participants had at least one year of experience using the Eclipse Java development environment; most had two or more years. Each participant participated in two or three sessions (Table 2). All participants in the study were initially newcomers to the ArgoUML code base. The participants worked on the same code base for each session and may have gained some familiarity with the code base over these sessions.

4 Study Results

We begin with a brief summary of the actions we observed participants perform and the tools they used. Following this summary we present the results of our grounded theory analysis in the form of eight observations (Table 3). These observations are based on the categories and dimensions we identified as being important to the situation under study. The description of each observation includes supporting data, often in the form of quotes from the recorded dialog.

The actions we observed may be summarized into five categories: (1) static exploration of the source code using a number of different tools, including viewing the source code in the editor; (2) setting break points and running the application in debug mode, which allows stepping through the execution and inspecting the execution stack and object values; (3) making paper notes; (4) making changes to the source code; (5) and reading online API documentation. Figure 2 shows the specific tools that were used during static exploration and debugging activities. The percentages in the figure represent the proportion of logged events (ignoring selections in the editor) over all sessions using the given tool. Various navigation actions, such as navigate to declaration, account for an additional 11.7% of the logged events and are not shown in the figure. This data indicates that the programmers did make use of many of the facilities of the development environment.

***Observation 1:* Goals were often decomposed into sub-goals that could be investigated directly, but the sub-goals were not always easy to form.**

In all of the sessions, participants broke goals into sub-goals that could be directly supported by the tools in the development environment. P4 described this process as *"trying to take [my] questions and filter those down to something meaningful where I could take a next step"* [S3]. As an example of this process, early in S10 the programmers determined that they needed to understand where in the code call events were being parsed. To accomplish this goal, four sub-goals were identified whose resolution were directly supported by the development environment.

1. *"Find out where [`MCallEvent`] gets created"* [P8]. The action for this sub-goal was a reference search on the specified class; this query returned two results which the participants inspected in the search results view. They decided that *"[`MFactoryImpl`] is a good place to start"* [P8].

2. Find out where the event object creation (`MFactoryImpl`) is initiated during the parsing of text input. From the search view, the programmers opened the `createCallEvent` method of the `MFactoryImpl` class and then searched for references to the method. They then followed a chain of three more searches for references directly in the search view, eventually resulting in them opening the `parseEvent` method from the `ParserDisplay` class in the editor.

3. Verify that the method found was in fact where the parsing takes place: *"do you think [`parseEvent`] is it?"* [P8]. To accomplish this sub-goal the participants set a break point and ran the application in debug mode. They were able to confirm their hypothesis when the breakpoint was hit.

4. Finally they wanted to find an appropriate point on the execution path to begin making changes to the code: *"so we have to search backwards from here"* [P8]. They used the variables view (in Eclipse's debug perspective) to determine various stages in the parsing.

Translating goals in this way was not always straightforward. For example, the question *"how does*

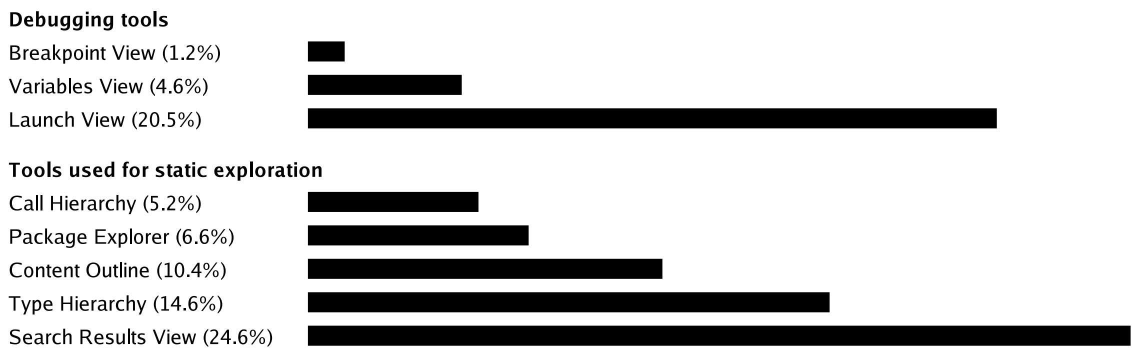


Figure 2. A summary of tool usage over all sessions. The percentages represent the proportion of recorded events using the given tool. Only those tools that accounted for at least 1% of the events are shown. Navigational events accounted for an additional 11.7% of the events.

`[MAssociation]` relate to `[FigAssociation]`?" [S3/P4] can not be answered directly in the development environment, but rather requires a significant amount of exploration to identify and integrate the various relationships involved.

Observation 2: **Goals were initially narrowly focused, but became more broad in scope as programmers struggled to understand the system sufficiently to perform the task.**

At the start of a change task, participants typically attempted to learn as little as possible about the system, focusing on very specific parts of the system, rather than learning about broader issues, such as the package structure or architecture of the system. For instance, the pairs in sessions involving task 1622 (S2, S4, S5 and S7) all focused on understanding how control reached the `PropPanelCallEvent` class to learn some key behavior assumed important for the task, rather than working to understand overall how the panel GUI worked.

For several sessions (most notably S6, S7, S9 and S12), the participants felt by the end of the session that to succeed with the task they needed to consider the system more broadly: *"I would definitely need a big picture, we are sort of in this small little bit of code [...] we need to back up further and see what else was out there"* [S12/P9]. This comment seems to have been caused by a realization that the solution for the task is more complex than first suspected. As another example, at the end of session S7, the driver felt that *"focusing on how to solve the task is too premature, because we're never going to figure it out if we are too narrow, I think we really have to get a wider view"* [P3]. The observer agreed that a broader view was needed, but felt that pursuing their current goals would lead them to that broader view: *"I think we just need to understand as much as necessary, and we're kind of circling in a spiral, realizing that we've got to get a bit further out"* [P1]. The strategy suggested by P1 here is consistent with a bottom-up approach to program comprehension as proposed in [1, 10], and more recently as part of the integrated meta-model [16].

Three other approaches were suggested by participants as a means of gaining a broader view: looking at architectural documentation, exploring the package structure, and using a *"brute force"* [S6/P2] approach, which involved stepping through the running application using the debugger, looking at much more of the system than had been considered up to that point.

Observation 3: **Programmers wrote code early in the change process.**

In half of the sessions (S1, S5, S7, S8, S10 and S12) programmers felt that a relatively narrow and incomplete understanding of the relevant code was sufficient to begin making changes to the source code. In these sessions, programmers began writing code as soon as they knew enough to begin. The pair in session S5 referred to their writing of code as *"mucking around with things"* [P3]. In contrast, in the other six sessions, the programmers set out to gain an understanding of all (or most) of the relevant code before beginning to make any changes. One programmer expressed discomfort coding too soon: *"I think I have just seen too many cases where [programmers] never come back and it is just really hard to maintain it afterwards"* [S8/P7].

When coding occurred it appeared to be part of an exploratory process that served several purposes. First, it minimized the amount of information that participants

Table 3. Summary of key observations.

1	Goals were often decomposed into sub-goals that could be investigated directly, but the sub-goals were not always easy to form.
2	Goals were initially narrowly focused, but became more broad in scope as programmers struggled to understand the system sufficiently to perform the task.
3	Programmers wrote code early in the change process.
4	Programmers minimized the amount of code that was investigated in detail.
5	Exploration activities were of two distinct types: (1) those aimed at finding initial focus points and (2) those aimed at building from such points.
6	Building a complete understanding of the relevant code was difficult.
7	Programmers' false assumptions about the system were at times left unchecked and made progress on the task difficult.
8	Revisiting entities and relationships was common, but not always straightforward.

needed to remember as many pieces of information that were learned could be captured explicitly in the code. Second, it served as a way to check assumptions, especially when combined with the use of Eclipse's debugger. For example, while writing code the programmers in session S5 kept the target application running in debug mode and continually switched between coding and inspecting the system in the debugger to ensure that the code they had written was executing when and how they expected it would. Finally, writing code helped support a narrow investigation of the system (Observation 2) as only those parts of the system needed to write the code had to be understood.

Observation 4: **Programmers minimized the amount of code that was investigated in detail.**

In addition to being narrow in their investigation of the code (Observation 2), the participants tended to avoid looking at source code in detail. For example, in S2 the observer said to the driver *"if I were you I would click on the interfaces, because the classes which implement it will have a lot of detail that is not so important"* [P1]. Similarly, the participants in S4 when considering the `MEvent` hierarchy (Figure 3) initially ignored the implementations of the classes and the interfaces and simply looked at the relationships involved, beginning with the type hierarchy.

In general, the participants appeared reluctant to look closely at the source code for an entity—a class or method—until after they had developed an initial understanding of the entity using tools that provided an abstract view of it, and until the participants felt that it was sufficiently important to their task. The participants in S3, were the most obvious exception to this rule as they, spent a relatively large amount of time reading source code, which appeared to be detrimental as little progress was made on the task. The more common and successful behavior we observed is consistent with the observation by Robertson et al. that programmers do not read source code line by line [8].

Also, programmers tended not to systematically explore more than about three search results. In some cases, rather than explore a large number of results the programmers would attempt to refine their query, although producing sufficiently precise queries was often difficult. In other cases the results were disregarded entirely.

Observation 5: **Exploration activities were of two distinct types: (1) those aimed at finding initial focus points and (2) those aimed at building from such points.**

Activities that focused on identifying relevant information can be divided into two sets: (1) those aimed at finding initial focus points and (2) those aimed at building from such points.

As an example of an effective approach to finding initial starting points, the programmers in sessions S5 searched for possibly relevant entities and set break points at some of these places. The application was then run in debug mode to see which of the identified points were in fact along the relevant control path. This approach gave the pair immediate feedback on their hypotheses, and confidence in what they had found. In session S9 the programmers began looking for such a point in the code by performing text searches based on an error message.

Given a relevant point (or points) to focus on, programmers often changed their approach and began building from that point, using several different means: references searches (*"let's see if `[targetChanged]` gets called"* [S6/P5]), opening the entity in the type hierarchy (*"what does `[NavPerspective]` inherit from?"* [S1/P3]), stepping through the method in the

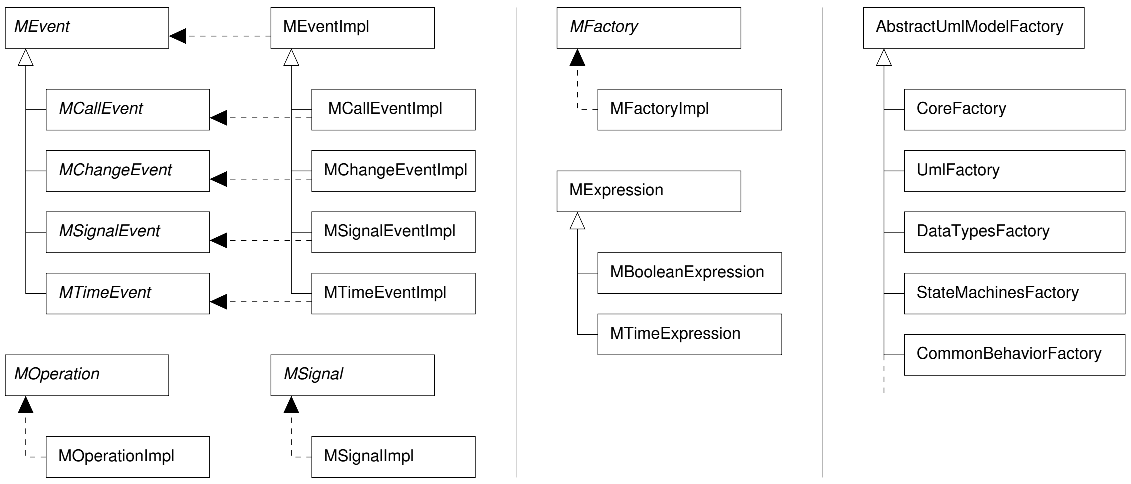


Figure 3. The model classes and interfaces (shown in italics) relevant to tasks 1622 and 1622a along with the factories for creating those elements.

debugger, and reading the source code. Similarly, Ko et al. report that once relevant code had been identified, programmers explored the code's dependencies [6].

Several times, programmers had difficulty identifying the information that they needed to perform the task. Sometimes the information could not be found because the tools were not helpful; other times the programmers did not make effective use of the tools. Tools were not helpful with data-flow issues and where the control-flow was obscured by the use of Java reflection; *"what was throwing me off was how much reflection was being used [...] the hardest part about reflection is that it just breaks the tools"* [S7/P3]. All of the participants that worked on task 1622 struggled with this issue.

Session S3 is an example where less effective use of tools hampered the process of identifying relevant entities. As mentioned above (Observation 4), the programmers in this session spent a significant amount of time reading source code in the editor and used Eclipse's other tools (searching and the debug perspective, for example) less often. They were also unsure of how the various searches worked and preferred to perform text searches and read the code. The programmers in this session never got to the point of having confidently identified initial focus points on which to build.

Observation 6: **Building a complete understanding of the relevant code was difficult.**

As entities and relationships were identified as relevant to the task and more information was discovered about those entities, an integrated model of how those entities fit together needed to be developed; *"I am kind of curious how [the `CoreFactory`] class integrates with this whole hierarchy"* [S2/P1]. In some cases the tools provided by the development environment acted as an external representation of the information to be integrated [20]. When the information was not or could not be externalized in this way, integration was often unsuccessful even when all (or most) of the relevant pieces of information had been correctly identified.

For both task 1622 and task 1622a understanding events and how they are created in ArgoUML was crucial. The key classes and interfaces involved are shown in Figure 3, though not all of the relationships are shown. Participants generally quickly identified the `MEvent` interface as important and opened it in the type hierarchy which externalized the information shown in the upper left corner of the diagram. The rest of the information necessary to understand ArgoUML events was identified but not externalized as a whole and none of the participants completely understood all of it.

To help build an integrated understanding, some participants drew structural diagrams on paper, presumably to take pressure off of their working memory. To explain why he was writing notes P1 said *"I know I am going to get lost"* [S2]. During the interview after the session on his use of paper and pen, a participant said: *"I was starting to forget who was calling what, especially because there is only one search panel at a time that I can see"* [S4/P6]. Even when paper was used, integration remained a difficult and important challenge.

Observation 7: **Programmers' false assumptions about the system were at times left unchecked and**

made progress on the task difficult.

An issue that impeded progress in several sessions (particularly S9, S11 and S12) was that of assumptions that were incorrect but never properly checked. These assumptions were only sometimes articulated explicitly. In session S9 the root cause of a false assumption appeared to be the misinterpretation of the condition (`!notContained.isEmpty()`) under which a particular exception was thrown; *"the hash table being empty is the problem, right?"* [P8]. This false assumption lead to other false assumptions and significant confusion. The programmers never identified and corrected their root error.

A common assumption was that the existing code in the system was correct. This fueled a desire to reuse system knowledge as observed by Flor et al. [3]. An example of this was observed while programmers worked on task 1622 and 1622a during which programmers attempted to use the partial implementation of call events as a guide for implementing other kinds of events. Although this approach was partially successful, the assumption that the existing code was correct was false and caused some problems. The code may have been incorrect because the ArgoUML code base is still in active development.

***Observation 8:* Revisiting entities and relationships was common, but not always straightforward.**

Much of the exploration we observed can be viewed as re-exploration. In fact 57% of observed visits to source code entities were revisits and, perhaps unsurprisingly, this proportion of revisits increased over time. In some cases this appeared to be because the discovered information had been forgotten, and in other cases because an earlier exploration had been stopped short or had been unsuccessful. This revisiting or re-exploration was sometimes intentional and other times not, and was sometimes noticed and sometimes not; *"we were retracing steps we had done before and [weren't] aware of it"* [S2/P1]; *"when we say let's look at that later, I think what that usually means is that if we come across this again using a different route, if there is an intersection somewhere, then we're going to look at this"* [S3/P4].

A number of different tools were used both for discovering new information and (later) for navigating back to that information. In session S5, for example, the programmers used Eclipse's inline type hierarchy both to find out about a type hierarchy and then to navigate between the members of that hierarchy. Similarly the search features of Eclipse were used to initially find a piece of information and then later to navigate back to it.

The activity of navigating was sometimes simple and direct; other times it was not. In a few rare cases, navigation was really difficult because the programmers had only a vague handle on the entity they wanted. In these cases, a certain amount of re-exploration was needed to complete the navigation. During session S6, the programmers wanted to navigate back to an entity they had visited one minute prior, but had difficulty doing so; *"the class disappeared"* [P5]. Eventually using the editor tab list, after about thirty seconds of effort, they re-found the entity of interest. At one point during session S11 the participants completely abandoned the effort to navigate back to a previously visited entity; *"do you remember in that stack trace of the exception where the method was that was sort of doing that save?"* [P2].

5 Study Implications

The participants in our study found the assigned tasks challenging. Many participants expressed a feeling of having made little progress during a session or specific parts of a session; *"waffled around"* [S2/P1]; *"we ended up going in circles for a long time"* [S6/P2]; *"we seemed to be spinning our wheels a bit"* [S11/P2]; *"I am not sure what we did in terms of progress"* [S12/P6]. In Section 5.1, based on our observations from the previous section, we highlight what we believe to be five important challenges the programmers faced in performing these tasks. In Section 5.2, building on our observations and the challenges identified, we present four suggestions for development tools that we believe may help programmers performing change tasks as newcomers.

5.1 Challenges

Gaining a sufficiently broad understanding. The participants struggled with gaining a sufficiently broad understanding of the relevant entities and relationships in the code base to correctly complete the task (see Observations 2, 5 and 6).

Cognitive overload. At times the amount of information that needed to be understood exceeded the amount that the programmers could manage mentally, causing cognitive overload [5]. To compensate, the programmers avoided looking at more detail than necessary and tried to minimize the amount of information that they attempted to understand (see Observations 2 and 4). Behaviors such as drawing diagrams and taking notes can be seen as an attempt to bridge local views to generate a more global understanding (see Observation 6) without relying on memory.

Navigation. Participants frequently navigated between entities that they had already visited (see Observation 8). Deline et al. believe that issues related to navigation and re-finding detract from programmers quickly accomplishing their tasks [2]. We hypothesize that when navigation is not direct additional cognitive effort is required of programmers, more pressure is put on working memory and developing a complete understanding is made more difficult.

Making and relying on false assumptions. Difficulties around unchecked false assumptions were presented in Observation 7. We believe participants made and failed to check false assumptions for at least three different reasons: (1) they misunderstood information they observed, (2) checking the information was difficult due to tool usage issues (see Observations 1 and 5) and (3) assumptions were not explicitly formulated as hypotheses that needed to be checked. Similar to Robillard et al., we found that relevant information is typically discovered only when deliberately searched for, which we believe allows false assumptions to continue [9].

Ineffective use of tools. Programmers varied in their use of tools. For example, some programmers made effective use of the debugger (see Observation 5) while others did not and as a consequence missed opportunities to directly discover information that they were trying to find. We also observed participants who did not understand the tools provided, such as being unsure of how to interpret search results (Observation 5), and who made less effective use of a particular tool, such as session S2 in which the participants could have used the type hierarchy viewer to externalize more relevant information.

5.2 Tool Implications

The goals of the participants and the success that they had in achieving those goals were heavily influenced by their use of available tools. Some of the issues about inadequate or inappropriate tool usage could possibly be remedied by training or by making the existing tools easier to learn and understand. We believe others can only be addressed by tools that more effectively support the activities we observed in our study. We present four suggestions of tools to help support some of these activities.

Using history. Tools that support a programmer in working with the history of their actions—for example, a notion of a visited entity or relationship—could be beneficial. This support could be an awareness of what has been visited, perhaps including some analysis to judge which entities and relationships are relevant and irrelevant. Support to easily navigate to entities previously visited might also be helpful.

Higher-level queries. Query tools that match the questions underlying a programmer's goals may help significantly. Several questions we observed programmers asking could not be easily answered with the tools they were using: how are two types related, how are two different object types handled differently by a set of methods, where a given object is accessed and how certain view elements map to model elements in a model-view framework.

Differentiating results. When a query returned a large number of results, such as a type hierarchy showing a large number of classes, the programmers were reluctant to investigate them and it was not always clear how to appropriately refine the query (see Observation 5). Some means of differentiating results, possibly by categorizing them, ordering (or emphasizing) them by relevance, or making the result view history aware, may help a programmer address this problem.

Synthesized views. Our results suggest that a view that shows more information simultaneously including multiple relationship types may help programmers build a more accurate and comprehensive mental model. Such a view may also help correct false assumptions.

6 Study Limitations

The use of pairs in our study likely impacted the change process we observed. We chose this approach to encourage a verbalization of thought processes and to gain insight into the intent of actions performed. An alternative approach to getting similar kinds of information is to have single participants think-aloud [15]. Of these two approaches, we believe, having participants work in pairs is more natural. Comparing the behavior we observed in our study with our own experience changing software systems and with the results of previous single programmer research studies (e.g., [6]) gives us a level of confidence that we have not greatly altered the process.

Our results are also impacted by the decision to choose change tasks that could not be completed within the allotted time, and tasks that involved modifications across multiple parts of the system. We chose complex tasks to stress realism and to stress the investigation of non-local unfamiliar code, a common task faced by newcomers to a system, and experienced by programmers working on changes that escape the immediate area of the code for which they have responsibility. We believe this trade-off was reasonable given the exploratory nature of our investigations. However, the implications we discuss must be interpreted in the context in which they were observed, and may not generalize to dissimilar situations.

7 Summary

In our observational study of how programmers manage change tasks we found that they were heavily influenced by the tools provided by the development environment. As programmers explored the source code looking for relevant source code entities and relationships they needed to take their goals and decompose them into sub-goals that could be directly supported by the available tools. This translation was not always straightforward and many goals required a significant amount of exploration using a range of tools. We also observed other challenges with identifying and understanding code relevant to the task.

Our findings suggest several potential research directions for development tools to better support the process of making changes to source code. Possible directions include: tools that leverage the history of the programmer's actions, tools that more closely match the questions a programmer needs to answer, different techniques for differentiating information such as search results, and views that synthesize more information for programmers.

8 Acknowledgements

The authors are grateful to Mik Kersten and Brian de Alwis for their valuable comments. This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] R. E. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[2] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of ACM 2005 Symposium on Software Visualization*, pages 183–192. ACM, 2005.

[3] N. V. Flor and E. L. Hutchins. Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance. In *Proceedings of the Empirical Studies of Programmers: Fourth Workshop*, pages 36–64. Ablex Publishing Corporation, 1991.

[4] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing, 1967.

[5] G. S. Halford, R. Baker, J. E. McCredden, and J. D. Bain. How many variables can humans process? *Psychological Science*, 16(1):70–76, January 2005.

[6] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 126–135. ACM Press, 2005.

[7] N. Miyake. Constructive interaction and the iterative process of understanding. *Cognitive Science*, 10:151–177, 1986.

[8] S. P. Robertson, E. F. Davis, K. Okabe, and D. Fitz-Randolf. Program comprehension beyond the line. In *Proceedings of the IFIP TC13 third international conference on human-computer interaction*, pages 959–963. ACM, 1990.

[9] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[10] B. Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers Inc., 1980.

[11] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: a case study of how software immigrants naturalize. In *Proceedings of the 20th International Conference on Software Engineering (ICSE)*, pages 361–370. IEEE Computer Society, 1998.

[12] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of CASCON'97*, pages 209–223, 1997.

[13] M.-A. Storey, F. Fracchia, and H. Muller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems, special issue on Program Comprehension*, 44(3):171–185, 1999.

[14] A. L. Strauss and J. Corbin. *Basics of Qualitative Research: Techniques and Procedures for developing Grounded Theory*. Sage Publications, 1998.

[15] M. W. van Someren, Y. F. Barnard, and J. A. Sandberg. *The Think Aloud Method; A Practical Guide to Modelling Cognitive Processes*. Academic Press, 1994.

[16] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 39–48. IEEE Computer Society, 1994.

[17] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, 1995.

[18] A. Walenstein. Theory-based cognitive support analysis of software comprehension tools. In *Proceedings of the International Workshop on Program Comprehension*, pages 75–84. IEEE Computer Society, 2002.

[19] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair-programming. *IEEE Software*, 17(4):19–25, 2000.

[20] J. Zhang. The nature of external representations in problem solving. *Cognitive Science*, 21(2):179–217, 1997.