Our paper reports on a qualitative study of programmer practices. The situation we studied was that of a programmer asked to perform a change task on code that they are unfamiliar with. 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, because we believed this understanding could have implications for the design of development tools.
Researches have studied situations more or less like this from various perspectives previously. Some work has focused on cognitive models of how a programmer comprehends a program. In comparison, we report on the broader process of how change tasks are managed, of which comprehension activities are just one piece.

More similar to our work are studies that qualitatively investigate work practices of programmers. We use some similar research techniques as those in [Flor 1991], but extend them to a larger pool of programmers working on more complex problems using today’s tools. As compared with [Singer 1997] we focus more closely on the activities directly involved in performing a change task. This gives a finer scale of analysis.

Some work based on more current development environments include two studies focusing on navigation. Our study differs from [Deline 2005 and Robillard 2004] in considering the change management process rather than focusing exclusively on navigation. And by [Ko] to elicit design requirements for a maintenance oriented development environment. We focus on a more realistic (or challenging) situation and aim to understand the reasons for the activities we observe.
9 programmers participated in 12 sessions. All participants were computer science graduate students with various amounts of previous programming experience. The range was from 10 years down to 1 year.

In each session the participants were given 45 minutes to work on a change task. This was followed by a 15 minute interview.

During each of these sessions participants worked as pairs. One participant (generally the less experienced programmer) was assigned to be the driver (i.e. given control of the keyboard and mouse). 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.
Tasks targeted ArgoUML, an open source UML modeling tool written in 60K lines of Java. 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.
To perform these change tasks participants used the Eclipse Java development environment which we take as representative of today's state-of-the-practice.

Here is a screen shot of it for those of you unfamiliar with Eclipse.
As I mentioned, this was an observational study. To structure our data collection and analysis in this exploratory study, we used a grounded theory approach which has is an emergent process intended to support the production of a theory that “fits” or “works” to explain a situation of interest. 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. Further analysis aims to organize and understand the relationships between the identified categories, possibly producing higher-level categories in the process.
Now I want to go through the results of our grounded theory analysis organized around 8 key observations. These observations are based on the categories and dimensions we identified as being important to the situation under study.
Observation 1:

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

“Trying to take [my] questions and filter those down to something meaningful where I could take a next step” (P4)

Questions

How does class A relate to class B?
How are objects of type A and type B handled differently by this collection of methods?
Where is this object accessed?
How do these view elements map to model elements?

Participants consistently broke their goals into sub-goals that could be directly supported by the tools in the development environment. One participant described this as “trying to take [my] questions and filter those down to something meaningful where I could take a next step”.

In some cases this was straightforward, other times not. For example here are generalized versions of questions asked by participants that could not be directly answered and instead required significant exploration. They needed to break these into many smaller questions and then piece together the answer. I will have more to say about this activity of piecing things together later, but first a bit more about goals.
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 one of the classes involved, rather than working to understand overall how the panel GUI worked.
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.

“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” (P9)

“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)

For several sessions, by the end of the session the participants felt that to succeed with the task they needed to consider the system more broadly. Some of this may have been caused by a realization that the solution for the task would be more complex than they first suspected. There was disagreement about the best way to get this broader view, but generally consensus that it was needed.
In half of the sessions the participants wrote some code. What was interesting to me and perhaps a bit surprising was that they did this with a relatively narrow and incomplete understanding of the relevant code. 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.

In these cases, coding appeared to be an exploratory activity and seemed serve several purposes: (1) it minimized the amount of information that had to be remembered because things could be immediately captured in the code, (2) it served as a way to check assumptions (especially when combined with the debugger), and (3) it served to support a narrow investigation of the code (only needed to investigate enough to write the next bit of code).
Observation 4:

Programmers minimized the amount of code that was investigated in detail

“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)

In addition to being narrow in their investigation of the code, the participants tended to avoid looking at source code in detail. For example on participant to the other: “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”.

Similarly participants in a another session initially ignored the implementation details of both the classes and the interfaces and simply looked at the relationships involved, beginning with the the type hierarchy. In general, the participants appeared reluctant to look closely at the source code for an entity (a class or method) until they had some initial understanding of it and believed it was sufficiently important to their task.
Participants seemed to have at least two distinct flavors of exploration. The first involved activities around finding some starting points, that is points in the code that were relevant to the task. The second involved activities around building on those points.

As an example of an effective approach to finding initial starting points, the programmers in one session 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 (if any) of the identified points were in fact along the relevant control path.

Given one or more relevant points to focus on, the participants often changed their approach and building from those points. These quotes are from those situations.

At times participants had problems finding the information that they needed to perform the task. Sometimes this was because the tools were not helpful and in a few cases because they didn’t make effective use of the tools. Examples of where the tools didn’t help include issues around Java reflection and data flow. See the last quote here for example.
As participants found and built on information that was relevant to the task (as I have just described), they needed to somehow put this information together. They needed to build and understanding that was both sufficiently broad (as I have already described) as well as complete.

Building this understanding was at times very difficult. That is a participant could have “seen” all the relevant information, but not completely understood it. Problems here described by participants as “getting lost”.

---

**Observation 6:**

Building a complete understanding of the relevant code was difficult

“I am kinda curious how [the CoreFactory] class integrates with this whole hierarchy...” (P1)

“I was starting to forget who was calling what, especially because there is only one search panel at a time that I can see” (P6)
The important thing to realize is that when you see information in pieces building an integrated understanding is difficult.

To help build an integrated understanding, some participants drew structural diagrams on paper, presumably to take pressure off of their working memory. Explaining why he used paper and pen, one participant explained “I was starting to forget who was calling what, especially because there is only one search panel at a time that I can see” (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

if (!notContained.isEmpty())
{
    throw new IncompleteXMIException();
}

“The hash table being empty is the problem right?” (P8)

Causes
• Misunderstanding information observed
• Checking the information was difficult due to tool issues
• Not explicitly formulated as hypotheses that needed to be checked

The issue of false assumptions (or maybe generally false information) about the software system reared its head at times for developers. This was particularly (though not only) an issue in 3 of the 12 sessions.

We believe participants made and failed to check false assumptions for at least three different reasons: (1) they misunderstood information they observed (this is an example here), (2) checking the information was difficult due to tool usage issues and (3) assumptions were not explicitly formulated as hypotheses that needed to be checked.

In the most serious cases, errors in understanding were layered on top of each other and making progress on the task almost impossible.
Our 8th and final observation. As participants navigated around the code base they often revisited entities and relationships. In fact 57% of observed visits were revisits and this proportion seemed to be increasing overtime. Some times this appeared to be intentional, other times not, as in this first quote. Several features in Eclipse were used to first find information and then later to navigate back to it (the inline type hierarchy view and the search view are two examples).

This activity of navigating between entities was sometimes simple and direct, but other times it was not. In a few cases it was really difficult and a certain amount of re-exploration was needed to complete the navigation. These last two quotes are from such situations.

We believe that revisiting is important to understanding, especially when things are in pieces and 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.
So I have presented 8 key observations that came out of the analysis of our results. Things like narrow versus broad exploration and issues around false assumptions.

In concluding I also want to mention some contributions that are in the paper, but not covered in this talk. Participants found the assigned tasks very challenging and in the paper we highlight 5 important challenges that impeded the progress the participants made on the tasks. These are closely related to the 8 key observations and include issues around cognitive overload and tool usage.

In the paper we also briefly explore the implications our results have for tool design, in particular suggesting 4 potential research directions that may help developers overcome these challenges they faced. Things like allowing a programmer to interact with their history or supporting queries that are closer to the questions programmers asked.

This has been a great start to a research project that is continuing. As an aside since running the study I have described in this talk we have run an interesting followup study which I would be happy to talk about off line if anyone is interested ...