An Industrial Case Study of Program Artifacts Viewed During Maintenance Tasks

Lijie Zou and Michael W. Godfrey
Software Architecture Group (SWAG)
School of Computer Science, University of Waterloo
{lzou, migod}@uwaterloo.ca

Abstract

Research on maintenance task structure modeling has so far examined only how often program artifacts are modified, and what information can be deduced from modification records. However, developers often access artifacts that they do not change, and this information is not modeled or recorded by current research systems. In this paper, we describe an exploratory industrial case study that we have conducted to investigate this issue; we found that within a given maintenance task, the software artifacts that are viewed but not changed outnumber the changed artifacts over 70% of the time. We further found that including information about which artifacts were changed and which were only viewed was key to a mature understanding of the tasks that the developers were performing. Finally, we discuss how creating a repository that captures both the viewed-only and modified artifact accesses can yield further insights into the development process, such as how developers handle interruptions and task switching in their workflow.

1. Introduction

Software maintenance is driven by *tasks*: clearly defined, goal-directed activities aimed at improving the software system in some way. When a developer is asked to fix a bug or add a new feature, (s)he will typically analyze code, reason about the design, and finally solve the problem. During this process, it is also typical that some program artifacts — such as source code files, documentation, or configuration files — will be changed to implement the solution, while others will be viewed for reasons such as program comprehension but left unchanged.

When a task is completed, changes to artifacts are usually recorded in version control systems and added to a repository. Recently, the research community has recognized that these software repositories contain significant latent knowledge about the development process, and has sought ways to "mine" them for insights into logical coupling [2], identifying expertise [6] and recommending task relevant information [11, 10, 7].

However, tracking which artifacts have changed and how captures only some of this latent task knowledge. Some knowledge — such as how to trace the symptoms of a particular bug or what artifact should be studied to best understand a given design decision — involves artifacts that have been viewed but not changed. In current research approaches, the uses of these viewed-only artifacts are not captured and modeled. Thus important task knowledge relating to them is lost.

We believe that by creating a new task structure model that includes both modified artifacts and viewed-only artifacts, richer knowledge about a task can be captured. If a historical repository of this augmented structure can be built, it will serve as a valuable source of information for the developer and manager. Improved understanding of the development process can then lead to the design of better processes and more natural supporting tools.

In this paper, we use an industrial case study to evaluate whether it is meaningful and important to capture the use of viewed-only artifacts in modeling maintenance tasks. Our initial results suggest that it is, and that the number of viewed-only program artifacts is often larger than the number of changed artifacts. Based on informal evaluation, our approach of building a repository that records both modified and viewed-only artifacts using instrumentation appears to be feasible. We have further found that the use of such a repository can yield new and unexpected insights into the software development process, such as the problem of frequent interruptions that developers may encounter while engaged in a maintenance task.

The remainder of the paper is organized as follows: first, we present our research questions in Section 2. Next, we describe the design of the case study in Section 3. In Section 4, we discuss the results obtained from this study, including the size of viewed-only program artifacts relative to modified program artifacts, and other other observations. In Section 5, we discuss related work. Finally, in Section 6, we summarize this study and describe future work.

2. Research questions

Our long term research goal is to build a task knowledge repository and use it to better understand and ultimately improve the software development process. We propose to include both viewed-only artifacts and modified artifacts as components of the task structure model. However, before that can happen, we must first study whether these viewed-only artifacts are sufficiently important to be included in the model. Specifically, we seek to answer two questions:

Q1: *Do viewed-only program artifacts exist?*

Q2: *What is the number of viewed-only program artifacts relative to the number of modified program artifacts?*

We are interested in these questions because if the number of viewed-only artifacts is insignificant, then obviously there is no need to continue with this line of research.

We operationalize the two research questions as follows:

We use $\#V$ and $\#M$ to denote the number of viewed-only program artifacts and the number of modified artifacts in a task respectively. To answer Q1, we check whether there exists a task that has $\#V > 0$. To answer Q2, we compare $\#V$ with $\#M$ in each task. We consider it reasonable to say that the number of viewed-only artifacts is big enough relative to the number of modified artifacts if following condition is satisfied:

$$\begin{cases} \#V/\#M \geq 50\%, & \text{if } \#M > 0, \\ \#V > \#M, & \text{if } \#M = 0. \end{cases} \quad (1)$$

In additional to answering the two questions about the existence and size of viewed-only artifacts, we also hope to understand their relevance to a task. Professional programmers are usually content to understand just enough to finish their maintenance tasks [9]. This implies that most viewed-only program artifacts should be relevant to the task at hand. Unfortunately, we lack detailed knowledge about *how* they are relevant. Are they mainly related to impact analysis, for example, or are they viewed to better understand a design decision? Answers to questions such as these are important to characterize task knowledge structure.

We have also other research questions that we hope to address in this study, including:

- What is the best way to build a repository that models information about both viewed-only and modified artifacts? We currently use a plug-in tool that captures events from Eclipse IDE. This tool requires some manual input from developer. Is this tool too intrusive for developers? What improvements can be made?

- What we can learn about software development activity from such a repository? Can we develop some sample applications?

3. Case study design

Our case study took place in a department, which we will label R, of a medium-size software company in Shanghai, China. We chose this company and department because it is available to us and has typical process of software development. For the study, we recruited three professional programmers from two project teams who use Eclipse as their major development environment.

We developed and installed a plug-in tool for their Eclipse environment that captures and records when program artifacts are being viewed or modified, and the developers used this as they were performing typical (and real) software development tasks. The study lasted for one month.

As stated above, our main goal of this case study is to evaluate whether it is important to include viewed-only program artifacts as part of the task knowledge. This is decomposed into two research questions as shown in Section 2. In additional to that, we also hoped to study the relevance of viewed-only artifacts to a task, evaluate the feasibility of building the repository, and develop sample applications for the repository.

For answering the two research questions about the existence and size of viewed-only program artifacts, the unit of analysis is a single task. For the other topics, the unit of analysis varies. Data is mainly collected using the Eclipse plug-in tool. Background information and data clarification are obtained using questionnaire, email, and informal meetings.

Data collected from the tool is loaded into a RDBMS and is further analyzed by querying the database.

3.1. Research setting

The software company involved in this study has about 300 employees. Its main product lines include finance, ERP (Enterprise Resource Planning), and business intelligence systems. The ERP department has passed CMM level 3, but the R department that was involved in our study has not been so certified. According to the R department manager, its software development practices have been strongly influenced by ERP department, and informally considers that they are similar in quality to them.

Three programmers — P1, P2 and P3 — from two project teams — H and B — participated in this study.

The H project is an internal application platform for several other major products in the company, including logistics, ERP, and finance. It has 577 classes and 57 KLOC, and is currently being maintained by six programmers. The B project is a business intelligence platform that provides advanced analysis of business data from other systems. One of its components is part of an open source project and the project team is contributing their enhancements back to the open source community. The B software system has 838 classes and 93 KLOC, if one includes the open source project, or 202 classes and 20 KLOC if it is excluded. Currently three programmers are involved in its development and the system is expected to be deployed within a month at time of writing.

3.2. Data collection

At the recruitment meeting, the prospective participants first filled out a questionnaire about their background and current work habits.[1] We found that all three programmers had joined the team shortly after the project was initiated. All of them are considered to be experts in their teams, have more than three years experience in programming in Java, and have used Eclipse for two years. All of them currently use Eclipse as their main integrated development environment (IDE). Their background seem somewhat similar. But this is not chosen intentionally. (Table 1 summarizes the details.)

A monitoring plug-in that we developed was then installed in the Eclipse environment they were using (the plug-in is described further in 3.2.1). The plug-in captures which program artifacts, such as Java classes or methods, were viewed or modified within Eclipse, and records this information into text files on the local machine. During the recruitment meeting, we gave a detailed tutorial on the use of the tool, and ensured that there were no unanswered questions.

[1] Prior to any data collection, all of the participants signed informed consent forms, as suggested by Office of Research Ethics at the University of Waterloo. All the raw data has been kept confidential and anonymity has been maintained.

Table 1. Background of participants

Prog	Proj	Years in proj	Years in Java	Years in Eclipse	Avg hours/day using Eclipse
P1	H	1.5	4	2	8
P2	B	0.5	4	2	5
P3	H	1.5	3	2	5

Monitoring lasted for one month. Data files were emailed to the researchers twice every week; this was done so that the developers could be reasonably certain that their manager was not tracking the data, which was a condition of the study.

More detailed information about how viewed-only program artifacts might be relevant to a task was collected through email. This was done one month after the monitoring was completed. A small number of tasks were picked and given to the programmer who performed them, with detailed information about what files were viewed only and what files were changed. Programmers were asked to recall why those viewed-only files were viewed. Were they viewed by accident, or due to their relevance to a task? In order to get enough information out of the small number of tasks, a relatively large number of viewed-only files were picked.

Some further background information and clarification of data was collected through email, and an informal meeting with the manager and the participants was held after the monitoring was completed to review how the study had gone.

3.2.1 The Eclipse plug-in tool

The Eclipse plug-in tool is the main data collection method used in this study. The tool captures events in Eclipse environment [1] and translate them into events that are meaningful for our study. For example, selection and editing events within editors are translated into viewing or modifying a program artifact. Currently, the tool works only within JDT, a Java development environment in Eclipse [4].

We consider three kinds of program artifacts: methods, classes, and files (compilation units). An event in a code segment is considered to be "belong" to the smallest program artifact surrounding the code: all the events within a method are considered as events of that method; all the events outside of any method but within a class are considered as events of that class; and all the events outside of any class but within a file are considered as events of that file. We choose the

smallest artifact to be the method since its is the basic functional unit of a program.

The plug-in tool has a view associated with it, as shown in Figure 1. In this view, a programmer can start or stop recording, start or stop a tasklet (a working unit of a task, this will be explained later), or view the events being captured in real time. The real time event viewing feature can be turned off to reduce performance overhead.

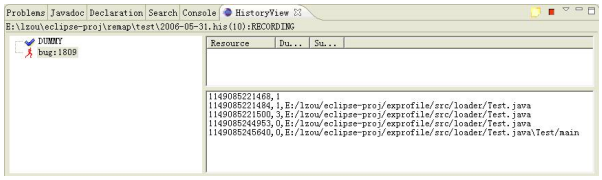

Figure 1. The Eclipse plug-in tool

Two important concepts in this study are the *task* and *tasklet*. A task is some work to be done with a defined goal, such as fixing a bug. The execution of two or more tasks may be interleaved. For example, a programmer may work on task A from 9:00 am to 10:00 am, then switch to high priority task B from 10:00 am until 11:00 am, and then go back to task A again from 11:00 am until 12:00 noon. In this example, switching to task B back and forth divides task A into two working units, 9:00 to 10:00 and 11:00 to 12:00. We define these continuous working units of a task as tasklets. In an IDE such as Eclipse, often a lot of information needs to be presented to the developer and many artifacts may be "open" at the same time. Task switching, therefore, causes extra work to save and recover task context.

Using our plug-in tool, a programmer can manually create a tasklet and specify the task that it belongs to. Tasklets with the same task name are regarded as different working units of the same task.

3.3. Data analysis

Data collected by the monitoring tool was loaded into a RDBMS, which made it convenient to query the data set from different angles. One programmer used two Eclipse instances at the same time for two days. Since our plug-in tool had errors capturing that (we had not foreseen this possibility), we did not include that chunk of data in our anaylsis.

Data analysis was performed both at the "method+" level and file level. In method+ level analysis, artifacts are of the granularity captured in the raw data. Since the raw granularity can be a method, class, or file, we denote it as the "method+" level. With file level analysis, data is lifted to the file level: a data point of a method or a class within a file is counted as a data point of this file. We choose a two level analysis because methods and files are two granularities that are commonly used in research; it is our hope that our results will be easy to compare with those of others.

3.4. Threats to validity

We now discuss the construct, internal, and external validity of our study.

- Construct validity

  Our study depends on the definition of *viewing a program artifact*. In our study, viewing a program artifact is determined by selection events in the Eclipse editor. This heuristic is not always correct for methods. For example, programmers may select somewhere within method A but then scroll the screen and look at method B instead without selecting on it. We plan to improve it in the future by considering the code visible on the screen. However, we note that this heuristic is always correct for files, since JDE only allows one file to be visible in the editor at one time.

- Internal validity

  Our work requires programmers to manually specify what tasks they are performing. If a programmer forgets to do so, then the tasks being captured will not match reality, thus adversely affecting internal validity. However, according to the participants, almost all of the time they were able to specify tasks correctly, so our study should be internally valid.

- External validity

  As with other case studies, our work may not be generalized to other industrial settings due to its particular context. Tasks in the two projects may have different characteristics than other projects. How these programmers understand and solve tasks may also be different from others. Replication of this study in other settings will be our future work.

4. Results

We will present our results in several parts. First we give an overview of the tasks performed in the study

Table 2. Overview of tasks

Prog	Days	#Tasks	#Tasklets	Hours using Eclipse
P1	25	20	38	165
P2	15	17	25	57
P3	7	6	11	98

Table 3. Type of tasks

Prog	#Tasks	Bug Fix	New Feature	Understanding	Others
P1	20	17	0	2	1
P2	17	7	7	3	0
P3	6	1	5	0	0

period. Then we show the results related to answering Q1 and Q2. In the following subsections, we discuss the relevance of viewed-only artifacts, the feasibility of building such a repository, and we present some insights about software development that came from studying the repository and case study results.

4.1. Overview

Table 2 gives an overview of the programmers' development activities during the study period.

P1 worked in Eclipse almost every weekday of the study period. P2 worked in Eclipse for half of the days; on those other days, P2 worked outside of the Eclipse environment, which is outside the scope of our study. P3 worked on the project for only the first eight days of the study; he switched to work on another project thereafter. A total of 43 tasks were performed during the study period. It is interesting to see that, on average, roughly one task was defined each programmer day.

Programmers manually specify the types of tasks they were performing. Table 3 summarizes this information.

The three programmers differed in the type of the tasks they most commonly performed. This difference in assignments was mainly due to the fact that they were responsible for different modules of the systems.

A task can be performed in separate working units (tasklets). Figure 2 shows the number of tasklets for each task in our study, sorted by programmer and starting time of each task.

We can see from the figure that most of the time, a task is completed within a single working unit. However, there also exist several occasions where a task was divided into several tasklets. About 20% tasks in this study has more than one tasklet. Separating one task into multiple tasklets may cause task switching. We will present more result of this topic in Section 4.6.1.

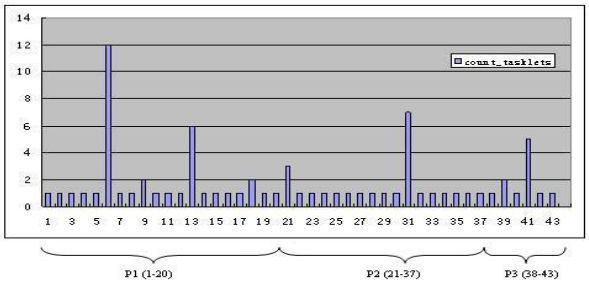

Figure 2. # Tasklets of each task

4.2. Q1: Do viewed-only program artifacts exist?

As we have explained in Section 2, we check whether viewed-only program artifacts exist by counting $\#V$, the number of viewed-only program artifacts in each task.

First we perform method+ level analysis. Figure 3 shows the number of viewed-only artifacts and the number of modified artifacts of each task at the method+ level, sorted by programmer and the starting time of each task (the same order as Figure 2). We display the number of changed program artifacts here to save space, since we will need both values in answering Q2. This is also true for Figure 4.

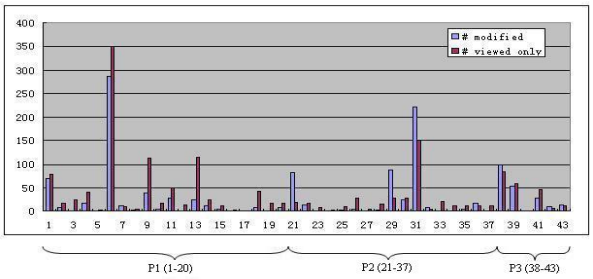

Figure 3. # Modified vs. # viewed-only at the method+ level

We can see from this figure that viewed-only program artifacts exist in almost all tasks (95%). Furthermore, some tasks consist only of viewed-only artifacts. For example, in one task that lasted for 40 minutes, all the 21 methods within four files were viewed only. These tasks were mainly for understanding purpose.

The 6^{th} task, which involved fixing a complicated bug, had the largest number of viewed-only artifacts; it was divided into twelve tasklets, as shown in Figure 2, with a total of 350 viewed-only artifacts and 286 changed artifacts at the method+ level. Some tasks had many more viewed-only artifacts than modified ar-

tifacts. For example, the 9^{th} task had 39 viewed-only artifacts versus 8 modified artifacts, and the 13^{th} had 114 viewed-only versus 24 modified.

We now perform analysis at the file level. Figure 4 shows the number of viewed-only files and the number of modified files for each task, also with the same order as Figure 2.

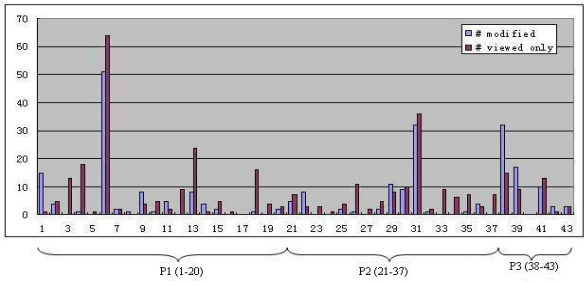

Figure 4. # Modified vs. # viewed-only at the file level

Similar to method+ level, almost all tasks (93%) included files that were viewed only. Also some tasks had many more viewed-only artifacts than modified, such as the 4^{th} and 13^{th} tasks.

Both results from the method+ level and the file level analysis have shown that viewed-only program artifacts do exist and they occur very often (more than 90% tasks in our study).

4.3. Q2: What is the number of viewed-only program artifacts relative to the number of modified program artifacts?

As we have explained before, we answer this question by comparing $\#V$, the number of viewed-only program artifacts, with $\#M$, the number of modified artifacts in each task.

The data from our study is shown in Figure 3 and Figure 4. We calculated $\#V$ and $\#M$ for each task both at the method+ level and the file level.

At the method+ level (Figure 3), the number of viewed-only program artifacts ($\#V$) is often larger than the number of modified program artifacts ($\#M$). In fact in our study, 79% (34/43) of the tasks have $\#V \geq \#M$. The average number of modified artifacts in a task is 27.5, and the median is 7. The average number of viewed-only artifacts is 35.6, and the median is 17. A total of 88% (38/43) of the tasks satisfy the condition (1), therefore their numbers of viewed-only program artifacts are considered to be big enough relative to their numbers of modified program artifacts.

At the file level (Figure 4), the number of viewed-only program artifacts is also often larger than the number of modified program artifacts. 74% (32/43) of the tasks have $\#V \geq \#M$. The average number of modified files in a task is 5.7, and the median is 2. The average number of viewed-only files in a task is 8.0, and the median is 4. A total of 81% tasks satisfy the condition (1).

So both the method+ level and the file level analysis show that in more than 70% tasks in this study, the number of viewed-only artifacts was larger than the number of modified artifacts. More than 80% tasks have the number of viewed-only program artifacts big enough relative to the number of modified program artifacts, as defined by the condition (1).

4.4. Relevance of viewed-only artifacts

We need information about how viewed-only artifacts can be related to a task in order to better understand the task knowledge structure.

Using the method we described in Section 3.2, we picked some medium size tasks that have a relatively large number of viewed-only files for each programmer, and asked them to comment on them. We selected the 4^{th}, 13^{th} and 18^{th} tasks for P1, the 26^{th} and 35^{th} tasks for P2 and the 41^{st} task for P3, from the sequence of tasks as shown in Figure 4.

Because we performed the review a month after the monitoring had been completed, we did not collect as much useful information as we had hoped for. P1 and P2 said that they could not recall the particular details any longer, but they still were able to describe some general situations that they considered to be factual. Only P3 described the details of the task we selected after he reviewed the code. Although this is not a good result for our data collection, it does serve to show how fast programmers can forget about what they did, and therefore how important it is capture this task knowledge while it is still fresh.

All the programmers mentioned that it is common to view but not change program artifacts. Sample scenarios include debugging, impact analysis, finding sample code to refer to, searching for reuse candidates, and general program comprehension.

They often consider whether a change to be made is compatible with the existing design and which solution is the best among all the possibilities. P3 described an example. There is an exception mechanism that involves a set of classes. When a new exception needs to be thrown, he needs to decide whether it can be defined as an instance of an existing exception class, or of a new subclass of an existing exception, or something

else. He said for the same reason of deciding how to throw an exception that best matches the existing exception design, several related classes are often viewed together.

There are also occasions when the artifacts that are viewed are not directly related to the task at hand. For example, it may suddenly occur to a programmer that some recently written code is incorrect. In such a case, he may examine that code while ostensibly within current task (i.e., without defining a new task, as he should). The programmers made several suggestions to improve this situation, such as automatic detection of task switching according to files being visited.

4.5. Building the repository

In our current approach, we use a plug-in tool that monitors activities within the Eclipse environment to collect raw data for the repository. The tool requires some manual input from programmers, and therefore its use may affect the normal work flow. To evaluate whether this is a problem to the programmers, we collected feedback from them through discussions and interviews.

Before the monitoring started, the programmers were worried about whether the tool would affect performance of Eclipse, considering that it captures many events within Eclipse. After using the tool for a week, they told us that the performance was not affected at all. They also mentioned that manually defining a task was not a big problem, since they usually perform a small number of tasks everyday.

They made several suggestions concerning what can be improved in our tool. One is creating an easier way to trigger the tool. Currently, all the functionality of our tool is accessed within a particular view. This view shares display space with other views in Eclipse, thus is invisible until programmers explicitly switches to it. The programmers suggested to use a shortcut, or a toolbar button instead to activate the view or pop up a new window. Another suggestion is to support automatic detection of task switching. One programmer said that in case of emergent task, he might forgot to specify the new task. This may be avoided if our tool can detect project switching without requiring the programmer to intervene.

4.6. Using the repository

After we built the repository, we analyzed its data from different perspectives. We observed several interesting phenomena related to software development.

4.6.1 Task switching

Task switching often occurs when programmers have other higher priority or easier tasks to complete. It may also happen when the current task is too complex, or does not have enough resources, thus needs to be delayed. In an IDE, it is common for developers to wish to have a lot of information within easy viewing. Task switching causes extra work to save and recover task context.

As a result of task switching, a task will be divided into separate working units (tasklets). Task switching will be seen as tasklets interleaved with each other. In our study, roughly 20% of the tasks have more than one tasklet, as shown in Figure 2 in Section 4.1. Three tasks had five or more tasklets, spanning three or more days. Some complex tasks, such as the 6^{th} task, had twelve tasklets that spanned eight days. When a task has multiple tasklets, it often interleaved with other tasks. For example, within the eight days of the 6^{th} task, six other tasks were interleaved. Another example is the 31^{st} task, which was divided into seven tasklets spanning six days, and mixing with three other tasks.

In summary, our study showed that task switching occurred fairly often in the two projects. It will be interesting to see whether this also happens in other project teams in this company, or in other industrial settings.

4.6.2 Interruptions

Our study also revealed some interesting aspects of work flow and interruptions. If there is no activity (i.e., an Eclipse editor event) detected by our plug-in tool for a period of time, then there is a good chance that the programmer has been interrupted, such as by a phone call or by taking a break. However, except by asking the programmer there is no reliable way of determining which pauses are interruptions and which are sessions of intent staring at a short piece of code (viewing a long piece of code requires mouse or keyboard actions, which would be caught by the tool as viewing events). However, our intuition and feedback from the programmers suggests that interruptions are the more likely occurrence.

There is a third case that must be considered. Interruptions detected by the IDE may not be interruptions at all; often programmers need to perform activities outside of the IDE to complete a task, such as examining a web page or accessing another tool that is external to the IDE. Again, we must rely on feedback from the programmers here: they confirmed that these "false positives" did indeed occur, but were relatively rare. They perform most of their work in Eclipse, except us-

ing another editor for editing xml files. Consequently, we decided to interpret IDE-indicated interruptions as genuine.

When interruptions occur in an IDE, the task context within the IDE that is being maintained by the programmer can be interrupted. When the programmer comes back to work on the task again, this task context needs to be recovered. If interruptions happen frequently, or if the task context is large, then recovering task context may require significant effort. There have been studies that try to help with task context management and recovery [5], but without empirical data about how interruptions occur in IDE, it is hard to evaluate the importance of task context recovery.

We now report some preliminary results relating to interruptions within IDE. More study and analysis will be performed in the future.

We must first decide what an interruption is and how it can be recognized. We use a threshold value to distinguish an artifact being viewed/modified from a likely interruption, such as a washroom break. We calculate the elapsed time between each pair of consecutive events. If the elapsed time is greater than a threshold value, we consider that an interruption has likely occurred. Otherwise, we use it as the duration of the first event.

Since there has been no study about how long a period of time in Eclipse without any event should be considered as an interruption, we tried different threshold values. Table 4 shows the total interruption time for each programmer if the threshold value is set to 5, 10, 30, and 60 minutes.

Table 4. Interruptions within Eclipse

Prog	Eclipse Time	#Hours / #Times of being interrupted			
		$\geq$ 5 min	$\geq$ 10 min	$\geq$ 30 min	$\geq$ 60 min
P1	165	92.5 / 290	75.5 / 145	48.4 / 48	28.7 / 18
P2	57	22.0 / 92	15.7 / 39	7.0 / 9	2.2 / 2
P3*	98	63.1 / 112	58.5 / 72	44.5 / 24	36.3 / 12

*: P3 performed two overnight tasks. If we do not consider the sleeping time as interruptions, then the number of hours of being interrupted on the last line should all be decreased by 16.4.

We can see from the table that many interruptions occurred while programmer are working in Eclipse. Roughly speaking, the average case is that there is a five minute interruption every half an hour, and a ten minute interruption every 50 minutes. Since lunch break is about one hour, the median frequency of interruptions will be higher than the average.

In our after-study meeting with the manager and the programmers, they admitted that they had noticed the interruption problem. The software systems maintained by the two teams are used by other teams within the company, and the programmers are often used as internal resources for expert advice and complaints. They are often interrupted by email or phone calls that ask them to fix bugs or explain what the software system does. Although short interruptions of five or ten minutes in our study may be due to normal development activities outside of IDE, the programmers expressed strong interest in a more detailed analysis of their interruptions and how they relate to their work patterns. This is an area of future research for us.

We have not studied what value should be set to the interruption threshold yet. Our informal experience suggests that five minutes is a reasonable value if programmers usually work actively within IDE. In the remainder of this section, we will use five minutes as the threshold value.

4.6.3 Time distribution within task

Knowing how much time was spent on viewing, modifying, or being interrupted, and how many interruptions occurred in each task, can help in understanding exactly how tasks were performed. Figure 5 and Figure 6 show this information from our study:

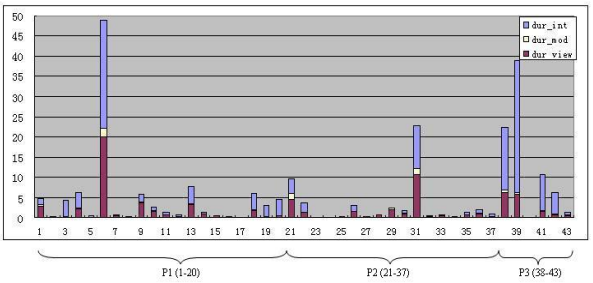

Figure 5. Time distribution

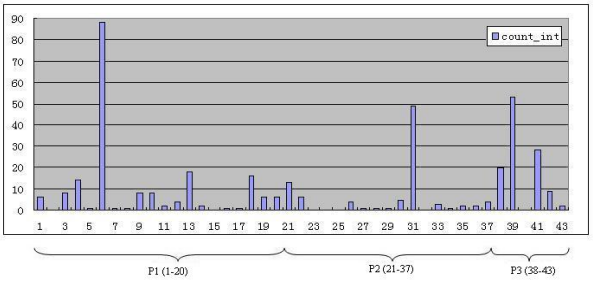

Figure 6. Number of interruptions

Figure 5 shows that in many tasks, interrupted time is larger than viewing time and modifying time. Large

Table 5. Viewing times and duration comparison

Method+ level				
	Modified		Viewed only	
Total Number	1183		1529	
	Median	Average	Median	Average
Times	10	34.8	2	4.6
Duration	34.3	186.8	8.2	42.2
File level				
	Modified		Viewed-only	
Total Number	246		343	
	Median	Average	Median	Average
Times	72	179.9	4	11.2
Duration	381.6	1012.1	24.4	105.8

tasks tend to have long interruption times. For example, the four longest tasks, the 6^{th}, 31^{st}, 38^{th} and 39^{th} tasks, have the four longest interruption times. As we can see by combining Figure 5 and Figure 6, these long interruption times are not due to a small number of long interruptions, but rather are due to many interruptions. For small tasks, some were not interrupted very much, such as the 14^{th} and 19^{th} tasks, while others had long interrupted time with many interruptions, such as task 20^{th}, 41^{st} and 42^{nd}.

If comparing Figure 5 and Figure 6, we can see that the two figures look very similar. This may suggest that the number of interruptions is related to the duration of a task. It may also suggest that some common interruption patterns exists among all of the tasks. Such pattern may be related to the organization dynamics or the social structure of the team and the company.

We can also see that the time spent on viewing is much larger than modifying. The median value of the ratio between viewing time and modifying time is 16. This conforms to the common notion that much of programmer's time is spent on program understanding rather than making changes.

4.6.4 Viewing times and duration comparison between viewed-only and modified

Information about how many times a program artifact was viewed and for how long can be important. Within a task, a program artifact that was viewed for twice may be less important than a program artifact that was viewed 20 times. Within a project, program artifacts that are often viewed by different programmers may indicate that they are the key artifacts of the software system.

Table 5 shows viewing times and duration for modified artifacts and viewed-only artifacts, at the method+ level and file level.

It is perhaps unsurprising to see that the changed artifacts were viewed more often and longer than viewed-only artifacts. When a piece of code needs to be changed, programmers will be more careful understanding what the code does and reasoning about its effect on other code. It is also interesting to see that the difference between modified and viewed-only artifacts becomes much bigger at the file level. For example, the median duration of modified vs. viewed-only at the method+ level is 4.2 (34.3/8.2). It becomes 15.6 (381.6/24.4) at the file level.

Based on the information about how many times and how long a program artifact was viewed, we have tried to find development "hot spots" within a project — that is, artifacts that are accessed frequently in a project. A development hot spot may indicate that it is a key component of a project, or it is hard to understand.

One file named `JDBCDataSession.java` in a development branch was found to be accessed in 11 tasks performed by P1. It was modified in six tasks, and was viewed only in the other five tasks. When we asked the manager why this file was accessed so often. He said that it is a kernel class of the project and therefore is relevant in many tasks. Anecdotally, we noted that this file has endured steady growth over time, and we wonder if it may soon need to be refactored.

5. Related work

5.1. Task structure

The work most related to our research is task structure as proposed by Murphy et al. [7]. A task structure is defined as "the parts of a software system and relationships between those parts that were changed to complete the task". The working version of task structure, task context, consists of parts and relationships of artifacts that are relevant to a developer as they work on the task. The relevance can be determined using different methods, such as the degree-of-interest(DOI) model [5] and structural dependency analysis [8].

Several types of applications can be built based on task structure, including recommending task relevant information, making IDE be more aware of the task context, sharing task structures within distributed team and forming a group memory.

Our goal is similar to this approach, that is to model knowledge relevant to understanding and solving a task, and use it to improve software development. The major difference is that we consider program artifacts that were viewed only as an important part of the knowledge model.

5.2. Mining software repository

A version control system that records historical changes can be used to develop techniques that help developers with their maintenance tasks. For example, it can be mined to identify logical coupling between program artifacts [2], recommend relevant files based on change patterns [11, 10], and locate experts within an organization [6].

The information recorded in version control system only consists of changes and brief comment describing each change. When reviewing these comments, other developers often still do not understand what happened in detail [3]. Actually, this is one of our motivations to includes both viewed-only and changed program artifacts in a repository. We hope by including more artifacts that were related to program understanding, developers can reduce their time understanding past changes.

6. Summary and future work

Program artifacts that are only viewed during maintenance tasks capture important knowledge about how a task can be understood and solved. However, current approaches only model and capture the artifacts that have been changed; artifacts that were viewed but not changed are ignored. Our empirical study has shown that that these view-only artifacts do exist and their number is larger than the number of changed artifacts in most instances, therefore justifying the importance of studying these viewed-only artifacts.

Informal data from this study also shows that our approach of creating a repository is feasible. Using the repository, we are able to observe many interesting phenomena in software development, such as task switching and interruptions within IDE.

We consider that the study we have presented in this paper, while fairly preliminary, suggests many possible avenues of future research. One idea we intend to explore is to more carefully characterize and model task knowledge structure. Another idea is to improve the plug-in tool we used to build the repository. We will also explore the creation of more applications based on such a repository.

7. Acknowledgments

We would like to thank BokeSoft Shanghai, China for providing the industry setting for this study. We would also like to thank the manager and all the participants in this study, for their precious time and valuable feedback. We would also like to thank the anonymous reviewers and Susan Sim, the WCRE-06 program co-chair, for their advice on the structure of this paper.

References

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

[2] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[3] R. E. Grinter. Using a configuration management tool to coordinate software development. In *Proceedings of the Conference on Organizational Computing Systems*, pages 168–177, Nov. 1995.

[4] JDT. http://www.eclipse.org/jdt.

[5] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 159–168, July 2005.

[6] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, New York, NY, USA, 2002. ACM Press.

[7] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Cubranic. The emergent structure of development tasks. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, July 2005.

[8] M. P. Robillard and G. C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 225–235, 2003.

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

[10] A. T. T. Ying, G. C. Murphy, R. T. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[11] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.