Activities Performed by Programmers While Using Framework Examples as a Guide

Reihane Boghrati
USC Database Lab
University of Southern California
boghrati@usc.edu

Abbas Heydarnoori
Department of Computer Engineering
Sharif University of Technology
heydarnoori@sharif.edu

Majeed Kazemitabaar
Department of Computer Engineering
Sharif University of Technology
kazemitabaar@ce.sharif.edu

ABSTRACT

It is now a common approach pursued by programmers to develop new software systems using Object-Oriented Application Frameworks such as Spring, Struts and, Eclipse. This improves the quality and the maintainability of the code. Furthermore, it reduces development cost and time. However, the main problem is that these frameworks usually have a complicated Application Programming Interface (API), and typically suffer from the lack of enough documentation and appropriate user manuals. To solve these problems, programmers often refer to existing sample applications of those frameworks to learn how to implement the desired functionality in their own code. This is called the *Monkey See, Monkey Do* rule in software engineering literature. The aim of this paper is to investigate and analyze the activities programmers perform to achieve a successful use of this rule. The results of this analysis will help us to build automated tools which are helpful for programmers while perusing the aforementioned Monkey See, Monkey Do rule.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Programming; Experimentation

Keywords

Object-oriented application frameworks; Program comprehension; Monkey See, Monkey Do rule

SAC'14 March 24-28, 2014, Gyeongju, Korea.

http://dx.doi.org/10.1145/2554850.2555016

1. INTRODUCTION

Software reuse has always been a concern in the field of software engineering. Software reuse results in several advantages, some of which are lowering the development costs, and increasing the quality and the performance of the system. Object-oriented application frameworks, such as Eclipse, would enable the reuse of both code and design [12]. In particular, frameworks provide reusable concepts, e.g., *context menu* on top of Eclipse, which are general units of functionality [8]. Programmers can then instantiate them in their own code via several implementation steps like sub-classing, implementing interfaces and, calling framework operations.

Frameworks provide a useful mechanism to reuse code in a specific problem domain [2]. Frameworks are a collection of abstract classes and their implementations. Programmers achieve their goals by adding their own codes and using the classes provided by frameworks. In other words, frameworks include abstract classes, their functionality, and their expectations of subclasses [4]. An important property of object-oriented application frameworks that distinguishes them from traditional software libraries is their *inversion of control* property. This means that frameworks often implement the main control loop of the applications and dispatch events that the application-specific handlers respond to. In other words, programmers write the subclasses of frameworks and pass the control flow to underlying frameworks [2].

The most important part of a framework is its Application Programming Interface (API). Understanding these APIs is typically complicated and hard, often due to their large size and the lack of enough user documentation [9]. To overcome these difficulties, programmers usually apply the *Monkey See, Monkey Do* rule [6]. The aim of this rule is to use existing sample applications of a particular concept in order to learn the implementation of that concept on top of the desired framework.

In this paper, our vision is to analyze the activities and behaviors of programmers while performing the Monkey See, Monkey Do rule. The results of this analysis will help software engineers to develop automated tools that would allow programmers to apply the Monkey See, Monkey Do rule, more efficiently and successfully.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes the experimental set up. Section 4 provides the results of our experiment. Section 5 discusses the results of the experiment and recommends some

solutions for a successful application of the Monkey See, Monkey Do rule. Section 6 provides the threats to the validity of the results. Finally, Section 7 concludes the paper and provides future research directions.

2. RELATED WORK

This work is related to multiple areas of research in software engineering and human-computer interaction (HCI). However, we consider the work in the following three areas which are of most relevance: (i) API usability; (ii) empirical studies of how programmers reason and learn; and (iii) empirical studies of code copying and reuse.

A lot of research has been done on cognitive aspects of programming and API usability. Robillard [10] indicates that APIs are often sophisticated to learn and there is a direct relationship between the API's *usage difficulty* and it's *learning difficulty*. Hence, when APIs become more complex, programmers mainly focus on learning only the parts that they need. Thus automated tools can be used to find source codes that could be helpful for programmers. Duala-Ekoko et al. [5] investigated the problems that arise when working with unknown or new APIs. To this aim, they observed 20 programmers while programming and asked them questions related to finding the appropriate packages that fulfill their programming tasks. Dagenias and Robillard [3] asserted the existence of many posts in forums and newsgroups for various APIs that unfortunately, after updating the framework APIs, their documentations are not updated. To mitigate this problem, they created a system that could recognize the required pseudo-code inside the API's documentation and then link it to related methods in the framework's API.

Hartmann et al. [7] observed programmers while searching the web for tutorials, libraries, sample codes, and documentations. Based on their observations, they developed a system called *Hypersource* that was able to preserve the relationship between a code snippet and its hosting webpage. Ying and Robillard [13] observed the activities performed by programmers while manipulating a code, and they concluded that programmers first need to comprehend the target code before actually modifying it.

Sillito et al. [11] analyzed and categorized the questions that programmers ask when modifying a source code. Brandt et al. [1] investigated the role of code snippets that could be found over the Internet on learning APIs. They showed that programmers use Internet when they are in an urgent need or when they are trying to recall the techniques they knew before.

3. THE EXPERIMENT

3.1 Objectives

As mentioned before, programmers often use existing sample applications as a guide when they want to implement new concepts on top of a desired framework. However, most of these samples may consist thousands of lines of code and comprehending such amounts of code can be a tedious task. Consequently, they need to find the relevant parts of the codes without getting involved in all the details. The goal of this study is to explore the ways that programmers may pursue to perform a successful use of the aforementioned Monkey See, Monkey Do rule. In other words, the most

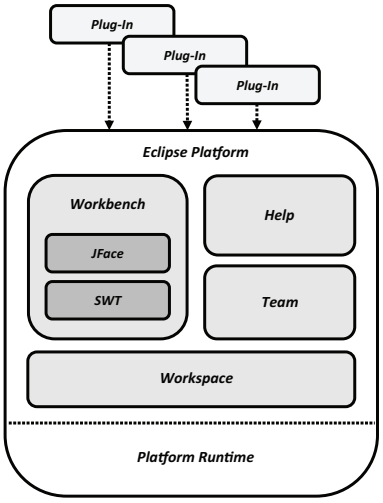

Figure 1: Eclipse Platform's Architecture

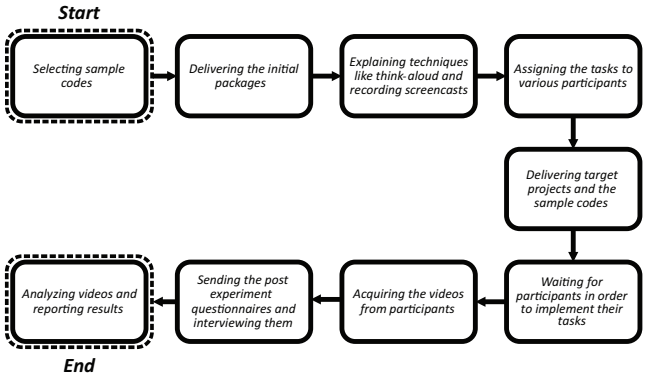

Figure 2: The steps of the experiment

significant question that this paper aims to answer is how programmers reuse previous codes to implement a desired concept. By analyzing the results of this study and understanding the methods that programmers employ, more efficient solutions can be designed to help them during applying the Monkey See, Monkey Do rule.

3.2 Experimental Design

In this section, we talk about the steps that we followed in designing this experiment (see Fig. 2). First and foremost, selecting a suitable framework on top of which participants implemented their code was an important decision to make. For this purpose, we chose the *Eclipse* framework. The Eclipse platform not only has the appropriate sophistication and comprehensiveness as an object oriented framework (e.g., see Fig. 1), it is an IDE (Integrated Development Environment) that many programmers are familiar with. Moreover, concepts can be implemented on top of this framework within a reasonable amount of time.

Next, since the concept *TableViewer* in the Eclipse platform has a moderate amount of sophistication that could be implemented in a meaningful time, we chose it as the desired concept. The *TableViewer* concept is an Eclipse view that has a tabular format, i.e., it has rows and columns. After

Table 1: A Summery of participants background

Programmer	Gender	Programming Languages	Semester	Programming Experience (Year)	Java Experience (Year)	Eclipse Experience (Year)
P_1	Female	C, C++, Java, PHP, Verilog, LISP, MATLAB	8	4	3	3
P_2	Female	C, C++, Java, MATLAB, Python	8	6	3	3
P_3	Male	C, C++, C#, Java, Python	8	4	1	1
P_4	Female	C, C++, Java, Verilog, Assembly, Pascal	6	5	3	3
P_5	Male	C, C++, C#, Java, MATLAB	8	6	1.5	1
P_6	Female	C++, Java, Python	8	6	3	3
P_7	Male	C, C++, Java, Python, LISP, Haskell, Basic, Pascal	10	10	4	3

selecting the desired concept, we created an empty Eclipse plug-in as the target project in which participants should implement their assigned concept, i.e., *TableViewer*. It is worth mentioning that we intentionally kept the size of this project small so that participants mainly focus on implementing the assigned concept instead of exploring the given target project. Then, we chose two Eclipse plug-ins as the example applications of the *TableViewer* concept.

Finally, we asked seven senior computer engineering undergraduates, referred to as P_1 to P_2 in the rest of this paper, to implement the *TableViewer* concept in the target project while following the Monkey See, Monkey Do rule. The participants were given some questionnaires to fill in order to give us some background information about their experience in programming. As shown in Table 1, the questionnaires indicated that all participants were familiar with Java and Eclipse to a high extent.

3.3 Running the Experiment

At this stage, we gave the participants the following items:

- *Documentation*: In this one page documentation, besides explaining the study goals to the participant, all the steps that he or she should take in order to complete the given task were also precisely described.
- *Background Questionnaire*: As its name implies, it asks about the participant's background in programming languages such as experience in Java and familiarity with the Eclipse platform.

To record the participants' behaviors, we introduced participants the *think-aloud* technique in which whenever in the experiment they perform an action, they should loudly state the reason of doing that. For this purpose, we asked them to use the *Camtasia* tool to record the required screen-casts. Fig. 2 illustrates other steps that were taken during the experiment.

4. EXPERIMENT RESULTS

In this experiment, we focused on analyzing what actions participants performed in order to fulfill their given task. Also we were curious about the effect of sample applications on the participants' procedures to accomplish the goal.

Among the seven participants that implemented the *TableViewer* concept, four of them could successfully finish their task while three of them failed to do so, although all of them were experienced Java and Eclipse programmers. Table 2 shows the time spent by each of the programmers to accomplish their tasks. All of those who failed to do the task mentioned that the most important reason was their lack of knowledge. In the final questionnaire, the participants were brought about the following two basic questions that provide us information regarding their attitude while following the Monkey See, Monkey Do rule:

1. What approaches are followed when facing the challenge of implementing a new concept?
2. What are the characteristics of a good sample?

Table 3 and Table 4 respectively indicate the answers of different programmers to the above questions. In the following, we will have a more detailed discussion about their answers to these questions.

Table 2: A Summery of participants results

Programmer	Time Spent	Success	Reason of Failure
P_1	45 minutes	✓	-
P_2	13 minutes	✓	-
P_3	40 minutes	✓	-
P_4	60 minutes	×	insufficient knowledge
P_5	60 minutes	×	insufficient knowledge
P_6	30 minutes	×	insufficient knowledge
P_7	10 minutes	✓	-

Table 3: The approaches that are taken by different participants when facing the challenge of implementing a new concept

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Searching Internet	×	×	×	×	×	×	×
Reading documents	×		×			×	
Designing Algorithms			×				×
Reading papers							×

4.1 Participants' Approaches

In the following, we provide a detailed description about the approaches that different participants followed to perform their assigned task while following the Monkey See, Monkey Do rule. A summarized view of these approaches and the programmers' activities is provided in Table 5.

Table 4: The participants' opinions about the characteristics of a good sample

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Smaller samples	×				×	×	
Abstract samples		×					
Being easy to follow				×			
Relevant comments			×			×	×

- Programmer P_1: She searched both sample applications for words similar to *TableViewer* to find something relevant. She copied some code that singly contained some errors in the target project. Thus, in order to eliminate the errors she copied codes that were dependent on it. Also, she used Eclipse's suggestions to solve the errors. She used the second project to find some similar parts. However, she could not eliminate the errors. She erased every thing and tried once more. This time, wherever she would encounter errors that seemed irrelevant, she deleted that line of code. She finally was able to successfully implement the given task.

- Programmer P_2: Since the concept *TableViewer* can be seen in the user interface, she went through the *UI* package and searched classes and methods in which the term *table* appeared. She copied relevant methods and to solve the errors, she either copied more code or deleted irrelevant parts of the code. Finally, this approach led her to success.

- Programmer P_3: Originally, he searched for the *TableViewer* concept in the first sample application and copied relevant methods and variables. However, there were several errors. So, in the next time, he copied relevant methods line by line and eliminated the parts he thought had nothing to do with his given task. This time he could successfully finish his task.

- Programmer P_4: Without any search, she started to figure out how exactly the code works and it took her a long time. She found the relevant methods and copied them, and in order to solve the errors, she copied more methods, variables and imported packages. Though there still existed some errors, she removed some packages which had caused the errors but did not change anything inside the methods, since she thought everything in the methods were only dependent on her written code. However, even after the removal of the packages, the errors still existed. She tried to solve the errors, but unfortunately, after spending one hour and seeing no sign of success, she gave up.

- Programmer P_5: At first, he searched for the terms *table* and *TableViewer* in the second project. Then he searched for some irrelevant words that did not help him. He switched to the first project and found the relevant methods. To solve the problems he searched for comments to understand the exact role of these methods and the codes inside them. He copied more methods and imported more packages to solve the errors. Afterwards, he also evicted some lines of code that had caused some errors. He continued copying more and more methods to solve the problems which ended up in adding more complexity to the code and he give up at last.

- Programmer P_6: She read the class names and found the relevant class, and copied some relevant methods. To solve the errors, she copied more methods. Though there still existed some errors, she erased all of the code and began again. This time she copied all of the methods of the relevant class and by deleting some methods and some lines which had errors, she tried to implement the given task but she was not successful at the end.

- Programmer P_7: He took a look at the first project and went through the *UI* package and found the relevant class. He copied two relevant methods but was encountered with some errors. So, he deleted both of the methods he had implemented and tried to find another way such as calling the sample application methods directly but he was not successful. Again he copied relevant methods and to solve the errors, he found out that some parts of these methods are not related to his goal. Hence, he deleted them and ran the project successfully.

4.2 Qualitative Analysis

Except programmers P_1 and P_5 who used both of the sample applications, the rest of the subjects only used the first sample application. It seems that using more than one example causes programmers to become confused. Most programmers used the first sample because it included the word *table* in the name of a class in which the *TableViewer* methods were implemented. This led programmers to readily find their desired methods. Moreover, it had a method named `createTable` that was creating a table as well as a method named `creatColumns` that was creating columns. On the other hand, the second sample was more complicated and suffered from unclear names. Furthermore, it had a class named `EditorListViewer` and had a method named `handleTableProviderChanged` in which a table was created as an event. Based on these observations, it seems that programmers perform the Monkey See, Monkey Do rule much better when appropriate naming conventions are followed.

Using both of the samples led programmer P_1 to become confused, while using one of the samples helped her to succeed. Also programmer P_5 tried to use both samples. Nevertheless, at the end, he did not successfully complete the given task. Using two samples, programmers have to struggle with the complexity of both of them and this reduces their concentration on accomplishing their goal. Moreover, searching both samples takes a lot of time and gives the programmers the sense that they are not able to finish the given task.

Programmers P_2 and P_7 who successfully implemented the task spent less than 15 minutes, while programmers P_4 and P_5 who failed accomplishing their task, spent almost 60 minutes. This shows that spending more time on a task does not necessarily lead programmers to a successful implementation and even makes them more confused in some cases.

Generally speaking, programmers who were brave enough to touch the code they copied were more successful. One of the programmers, i.e., P_5, who did not change anything in

Table 5: Programmers' activities while performing the Monkey See, Monkey Do rule

Activity	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Searching the code for words similar to the assigned concept	×	×	×	×	×	×	×
Eliminating reference errors by copying methods	×	×	×	×	×	×	×
Copying methods that seem relevant	×	×	×	×	×	×	×
Removing code lines that seem irrelevant or have errors	×	×	×	×	×		×
Getting rid of errors by copying variable declarations	×	×		×	×	×	×
Searching comments for words similar to the assigned concept	×	×	×			×	
Adding or modifying code			×				×
Using Eclipse's suggestions to debug syntax errors	×				×		
Learning the code inside of each method				×			
Fully or partially copying sample code's packages				×		×	
Inserting comments in the target project		×					
Copying from sample codes line by line			×				
Leaving methods untouched				×			
Copying plug-in dependencies				×			
Searching for specific similar comments					×		
Copying and adding a great number of methods					×		

the copied methods (either add or delete) was not successful. Moreover, the programmers who were successful did not struggle to understand all the details of the sample applications' code. More specifically, all the programmers were seeking to find relevant names in class names or method names. Furthermore, four of them were searching for relevant names in comments. Based on these observations, the most important factor of a sample's usefulness is the relevancy of names and comments.

The approach mostly followed by programmers to perform the Monkey See, Monkey Do rule was mainly copying code from the sample applications. However, two of the programmers did some coding as well although is was just defining some new variables. When copying the code, some errors came up and to solve them, programmers mainly followed two approaches: copying more codes that seemed relevant, or deleting the erroneous parts. As mentioned before whoever preferred deleting rather than copying more and more codes were more successful. As noted before, there were three programmers, i.e., P_4-P_6, who copied the whole or most of a package from the sample application, and they were not successful at the end. In other words, it just made them to become more confused. Consequently, programmers have to be careful about what they decide to copy.

One of the programmers, i.e., P_3, started by copying the code line by line from the sample application. So, he could keep track of what is related to his task and finally he succeed to implement the given task; however, it took him about 40 minutes. There was also a programmer, i.e., P_1, who put comments in her project to see how it works and this helped her to succeed finally.

Programmer P_5 spent two much time to figure out what exactly the samples do, but he did not succeed. It shows that there is no need to understand the whole sample and it is somehow waste of time.

Tables 3-4 show programmers' answers to the final questionnaire. As these tables indicate, two programmers mentioned that the problem with the samples was their big size and three of them asserted that smaller samples will be more useful. Two of them believed that relevant names was very useful and three of them expressed the more relevant comments exist, the more samples are useful. All of the programmers express that they will use the Internet when they encounter new concepts, either reading forums or using existing samples. Thus, web pages are an essential source to help programmers and if sufficient documents and relevant comments are provided to explain how various elements of the program operate, developers will implement their tasks much faster and more successfully.

Two of the programmers mentioned that they will read documentation while encountering new concepts. Also two others stated that they will put some time to think about an algorithm to implement the given concept. One of them also asserted that he would read related papers if needed.

One of the programmers believed that a sample application could be characterized as good if it provides the implementation of the desired concept in an abstract way. Another programmer mentioned that it would have been beneficial for developers if there was a way to follow the code line by line.

5. DISCUSSION

The observations provided in Section 4 give us a general insight of what programmers do and what steps they follow to have a successful Monkey See, Monkey Do process. This information can lead us to find out ways that enable us to help them in applying this rule. In particular, some possible approaches are:

- Recommend more practical samples in order to help programmers develop their applications not only in less time but also with higher quality. Practicality can be assured by the convenient use of variable/method names and clarity in comments.

- To increase the efficiency of programmers, some automated tools could be developed that can assist programmers in applying the rule.

Our experiment showed us that if we were able to keep track of all the concepts in a sample application (e.g., by using a separate XML document) that is able to annotate each class, method, or even a single line of code with a specific

tag that shows each coding element belongs to which concept, then programmers can easily apply the Monkey See, Monkey Do rule.

Furthermore, the coding elements that are suggested to a developer could be represented as a hierarchy in which the most important methods would be shown at higher levels. This seems to substantially increase the programmers' knowledge on how to properly differentiate between application-dependent code (e.g., the line of code that creates a *Table*) and concept-dependent code (e.g., the code lines that determine the properties and attributes of the *Table*), which was one of the main difficulties that our subjects struggled with in this study. This would enable us to create an online repository of open-source sample applications.

6. THREATS TO VALIDITY

In the following, we discuss three of the most important threats to the validity of the results of the experiment presented in this paper:

- *Participants:* In order to be able to generalize the results of this experiment, participants should be chosen from various levels of expertise and familiarity with programming languages and frameworks. Moreover, the number of participants will directly influence the results. To address these issues, in our future studies we plan to run the experiment with more subjects and of a wider range of expertise.

- *Framework and Concept:* One potential threat to the validity of the results of this experiment is that it is done by using only one framework, i.e., Eclipse, and one concept, i.e., *TableViewer*. However, Eclipse is a popular framework which is widely used in practice. Furthermore, the concept *TableViewer* is a moderately difficult concept which needs to be implemented in many of the Eclipse plug-ins. Nevertheless, it is still necessary to perform the experiment with more frameworks and concepts with various characteristics. This is left for future work.

- *Replicability:* The setup of this empirical study is discussed in detail, as well as the data collection and analysis methods. The sample applications are open source and available upon request. Consequently, it should be possible to replicate the experiment.

7. CONCLUSIONS AND FUTURE WORK

This paper presented the results of an experiment done to empirically study and analyze the programmers' activities while performing the Monkey See, Monkey Do rule, i.e., use existing example applications of a framework as a guide to develop new applications on top of it. Briefly speaking, we had the following observations in our study: (i) programmers often copy the code that seems relevant from the sample applications into their own code and then modify it to get the desired result; (ii) having meaningful names and comments is a helpful tool for programmers to find the implementation of their desired functionality in sample applications; (iii) there is no relation between the time people spend on a task and their chance of success; (iv) spending too much time on figuring out what the sample application is actually doing does not guarantee success; and (v) programmers typically refer to Internet whenever they face the challenge of implementing new concepts.

In future, we plan to perform the experiment with a larger number of programmers and concepts from various frameworks in order to get more reliable results.

8. REFERENCES

[1] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of the SIGHCI Conference on Human Factors in Computing Systems*. ACM, 2009.

[2] G. Butler. Object-oriented frameworks. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001.

[3] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 2012.

[4] L. P. Deutsch. Reusability in the smalltalk-80 programming system. *IEEE Tutorial on Software Reusability*, 1987.

[5] E. Duala-Ekoko and M. P. Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the 2012 International Conference on Software Engineering*. IEEE, 2012.

[6] E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley Professional, 2004.

[7] B. Hartmann, M. Dhillon, and M. K. Chan. HyperSource: Bridging the gap between source and code-related web sites. In *Proceedings of the SIGHCI Conference on Human Factors in Computing Systems*. ACM, 2011.

[8] A. Heydarnoori, K. Czarnecki, W. Binder, and T. T. Bartolomei. Two studies of framework-usage templates extracted from dynamic traces. *IEEE Transactions on Software Engineering*, 38(6):1464–1487, 2012.

[9] D. Hou, J. Hoover, and C. Yin. The framework use problem: A preliminary study with GUI frameworks. In *Proceedings of the 1st Midwest Software Engineering Conference*, 2003.

[10] M. P. Robillard. What makes APIs hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[11] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[12] J. Viljamaa. Reverse engineering framework reuse interfaces. *ACM SIGSOFT Software Engineering Notes*, 28(5):217–226, 2003.

[13] A. T. Ying and M. P. Robillard. The influence of the task on programmer behaviour. In *Proceedings of the 19th International Conference on Program Comprehension*. IEEE, 2011.