Comprehending Implementation Recipes of Framework-Provided Concepts through Dynamic Analysis

Abbas Heydarnoori
David R. Cheriton School of Computer Science,
University of Waterloo, Canada
aheydarnoori@uwaterloo.ca

Krzysztof Czarnecki
Electrical and Computer Engineering Department,
University of Waterloo, Canada
czarnecki@acm.org

Abstract

Application developers often use example applications as a guide to learn how to implement a framework-provided concept. To ease applying this technique, we present a novel framework comprehension technique called *FUDA*. FUDA integrates a new dynamic slicing approach with clustering and data mining techniques to generate the implementation recipes of a desired concept.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—reverse engineering; D.2.13 [*Software Engineering*]: Reusable Software—reusable libraries.

General Terms Design, Documentation, Experimentation.

Keywords Object-Oriented Software Frameworks, Framework Comprehension, FUDA, Dynamic Slicing, Clustering, Data Mining.

1. Introduction

Object-oriented software frameworks have shown to be one of the most important reuse technologies available today by capturing the commonalities of an application domain into a set of carefully designed abstract classes with well-defined collaborations. However, the Application Programming Interfaces (APIs) of many modern frameworks are complex and difficult to learn. To cope with this problem, application developers often use existing framework applications as a guide to understand how to implement a specific framework-provided concept (e.g., a context menu in the Eclipse environment using the *JFace* framework). However, the main challenge of this approach is that the code implementing the desired concept often involves multiple classes, and may be scattered across and tangled with code implementing other concepts. Therefore, tools and techniques are required to help finding the relevant instructions among potentially many thousands lines of source code.

To address this challenge, a number of techniques have been proposed in literature. Examples include code searchers (e.g., [8, 5]) and static code miners (e.g., [7, 9]). These techniques apply static analysis on the source code of existing example applications and allow retrieving code fragments or usage rules for a particular API element. Although these techniques can be very helpful in situations where the developer at least knows the name of the API element of interest, they are less helpful if the developer has only a high-level idea of the concept that needs to be implemented. In the latter case, a concept location technique based on dynamic analysis (e.g., [2]) can be used to locate the code implementing the concept of interest. However, these techniques can only identify the appropriate calls from the application to the framework API and vice versa (i.e., framework API interaction) when the concept of interest is invoked. Unfortunately, implementing a framework-provided concept involves not only implementing those calls and callbacks, but also creating, connecting, and destroying the objects that are called when that concept is invoked. Since the object creation and set-up as well as the necessary clean-up code often happen well before and after the desired concept is invoked, the existing dynamic concept location techniques would miss them.

To address the above challenges in existing techniques for framework comprehension and concept location, we developed a novel framework comprehension technique called *FUDA* (*Framework API Understanding through Dynamic Analysis*) [3]. FUDA extracts the implementation recipes of a desired framework-provided concept from dynamic traces with the help of a dynamic slicing approach integrated with clustering and data mining techniques. The following section provides an overview of this approach.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-865-7/07/0010.

2. The FUDA Approach Overview

In FUDA, a framework-provided concept is defined as a functionality that is realized in the example application's source code by using the framework's API. Furthermore, the concept has to have a visible behavior that can be triggered by the user through the graphical or programmatic interface of the application. The basic idea of FUDA is to generate the implementation recipes for the desired framework-provided concept by using the information collected at runtime. For this purpose, a number of example applications implementing that concept are selected and are run according to a number of use cases invoking it. At runtime, the traces of all the interactions between the example applications and the framework API are recorded, and those ones that happen when the concept of interest is invoked are marked by the software engineer using a trace marker utility.

The collected dynamic traces are then sliced with respect to the marked interactions by applying a novel dynamic slicing approach introduced in [3]. The purpose of this step is to find and mark other interactions that might be relevant to the implementation of the desired concept and happen either before or after invoking it. This dynamic slicing approach operates on framework API interaction traces rather than on complete instruction traces as in the case of traditional dynamic slicing [1]. For this purpose, we introduce a new type of data dependence graph called *AIDG* (*Framework API Interaction Dependence Graph*). An AIDG is a directed graph that represents the potential runtime data dependencies among the events in the framework API interaction trace with respect to their order of execution.

After slicing the framework API interaction traces and marking the concept relevant interactions, they are analyzed in a batch-like manner to generate the concept implementation recipes. For this purpose, since it is typically possible to implement the same concept by using different framework-provided abstractions, the framework API interaction traces are clustered by a *hierarchical agglomerative clustering* algorithm [6] based on their constituent marked interactions. The aim of this phase is to explicitly let the user know different ways that her concept of interest can be implemented.

Following clustering the dynamic traces, *mining frequent closed itemsets* [10] data mining technique is applied on each cluster of dynamic traces separately to determine the concept implementation recipes. Frequent closed itemsets are the set of commands that are frequently used together in example applications to implement the concept of interest. Each concept implementation recipe specifies what framework API classes should be instantiated by the application, what methods should be implemented on the application's side, and what framework API methods should be called by the application.

3. Empirical Evaluation

To perform empirical evaluation, FUDA is prototyped in Java (SDK 5.0) as two separate but related Eclipse plug-ins [4]: *Trace Collector Plug-in* (*TCP*) to record and mark framework API interaction traces, and *Framework Comprehension Plug-in* (*FCP*) that applies the FUDA technique on the dynamic traces collected by the TCP.

We used the prototype implementations of FUDA to generate the implementation recipes for the following sample concepts: (1) a context menu in an Eclipse view, (2) an Eclipse view that supports tree viewers, (3) an Eclipse view that supports table viewers, and (4) drawing a figure in a GEF editor. The generated implementation recipes are then analyzed both quantitatively and qualitatively by comparing them against their corresponding framework API documentation. The quantitative analysis shows that it is possible to generate the implementation recipes for a concept of interest with a high precision and recall by applying the FUDA technique on only a few example applications. The qualitative analysis confirms that the generated implementation recipes are comparable to those found in the framework API documentation.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, New York, NY, USA, 1990. ACM Press.

[2] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE TSE*, 29(3):210–224, 2003.

[3] A. Heydarnoori and K. Czarnecki. Comprehending object-oriented software frameworks API through dynamic analysis. Technical Report CS-2007-18, University of Waterloo, Waterloo, ON, Canada, 2007.

[4] A. Heydarnoori and K. Czarnecki. Mining implementation recipes of framework-provided concepts in dynamic framework API interaction traces. In *OOPSLA Companion*, 2007.

[5] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, pages 117–125. ACM Press, 2005.

[6] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Survey*, 31(3):264–323, 1999.

[7] A. Michail. Data mining library reuse patterns using generalized association rules. In *ICSE*, pages 167–176. ACM Press, 2000.

[8] N. Sahavechaphan and K. Claypool. XSnippet: Mining for sample code. In *OOPSLA*, pages 413–430. ACM Press, 2006.

[9] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *MSR*, pages 54–57. ACM Press, 2006.

[10] M. J. Zaki. Mining non-redundant association rules. *Data Mining Knowledge Discovery*, 9(3):223–248, 2004.