Supporting Framework Use via Automatically Extracted Concept-Implementation Templates

by

Abbas Heydarnoori

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2009

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Object-oriented application frameworks allow the reuse of both software design and code and are one of the most effective reuse technologies available today. Frameworks provide *domain-specific concepts*, which are generic units of functionality. Framework-based applications are constructed by writing *completion code* that instantiates these concepts. The instantiation of such concepts requires implementation steps in the completion code, such as subclassing framework-provided classes, implementing interfaces, and calling appropriate framework services. Unfortunately, many existing frameworks are difficult to use because of their large and complex APIs and often incomplete user documentation. To cope with this problem, application developers often use existing framework applications as a guide. While existing applications contain valuable examples of concept implementation steps, locating them in the application code is often challenging.

To address this issue, this dissertation introduces the notion of *concept implementation templates*, which summarize the necessary concept implementation steps, and a technique named *FUDA* (*Framework API Understanding through Dynamic Analysis*) which automatically extracts such templates from runtime information collected when that concept is invoked in two or more different contexts in one or more sample applications. The experimental evaluation of FUDA with twelve realistic concepts on top of four widely-used frameworks suggests that the technique is effective in producing quality implementation templates for a given concept with high precision and recall from only two sample sample applications and execution scenarios. Moreover, it was observed in a user study with twelve subjects that the choice of templates vs. framework documentation had much less impact on development time than the concept complexity.

Acknowledgements

This is an opportunity for me to express my gratitude towards everyone who has supported me throughout the course of this PhD. First and foremost, I would like to sincerely thank my supervisor, Professor Krzysztof Czarnecki, for his encouragement, guidance, and support. His vast knowledge and experience led my research towards the right direction and helped me overcome various obstacles. I am also grateful for his great dedication and compassion that made him a valuable supervisor.

I would like to thank all the members of my dissertation committee: Professor Uwe Aβmann, my external examiner, for managing to travel all the way from Germany to attend my oral defense despite his busy schedule, and provide me his insightful comments; Professor Mike Godfrey and Professor Joanne Atlee for taking the time and the effort to participate in my committee and provide me their thoughtful feedbacks on my work along the way; Professor Sebastian Fischmeister for his invaluable remarks on the thesis; and Professor Patrick Lam for accepting to serve as a delegate in my oral defense. I also would like to thank Professor Farhad Mavaddat who supervised the early stages of my doctoral research. Many thanks also go to Professor Farhad Arbab who provided me the opportunity to visit Centrum voor Wiskunde en Informatica (CWI) in Netherlands in Spring 2004.

A big thanks to all my friends at the University of Waterloo, especially my friends at the Generative Software Development Lab. I was extremely fortunate to spend enjoyable time with some great lab mates. In particular, I express my appreciation to Thiago Tonelli Bartolomei for his contribution to this research and Mohamed Abd-El-Razik, Michal Antkiewicz, Ca Bol Chan, Reza Dorrigiv, Afshar Ganjali, Allen George, Chang Hwan Peter Kim, Herman Lee, Rafael Lotufo, Jason Medeiros, Marcilio Mendonca, Ali Razavi, Steven She, Hossein Sheikh-Attar, and Matthew Stephan for their kind help.

While living in Waterloo, I and my wife have been fortunate to have a warm community of Iranian friends who made Waterloo a pleasant environment for us to live and study. My special thanks go to all of them.

I am deeply thankful to my dear parents, Javad and Mehri, and my brothers and sister, for their belief in me and their support throughout all of my life in every possible way. I can not imagine how I could have achieved success in my life without their prayers, help, and encouragement. I am also sincerely thankful to my parents-in-law, Farsad and Zohreh, and my sisters-in-law for their prayers, kindness, and nice wishes.

Last, but most certainly not the least, my profound appreciation and love goes to my beloved wife, Mahsa. There is no doubt in my mind that this work would not have been accomplished without her help and support. I really appreciate her encouragement and patience and all the delight and happiness she has brought into my life. My love and passion for her go far beyond the expressive power of words.

Dedicated To:

My beloved wife, Mahsa,

for her love, patience, and enthusiastic support.

My dear parents, Javad and Mehri,

for their countless sacrifices and making me the person I am today.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Software reuse is one of the important goals in software engineering that can improve the quality and productivity of software development. To this aim, *object-oriented application frameworks* have shown to be effective since they allow the reuse of both design and code [125]. According to Gamma et al. [39], an object-oriented application framework is "a set of cooperating classes that makes up a reusable design for a specific class of software". In other words, by capturing the commonalities of an application domain into a set of carefully designed abstract classes with well-defined collaborations, frameworks enable reuse at both code level and design level.

Frameworks provide *domain-specific concepts*, which are generic units of functionality. For example, a graphical user interface (GUI) framework such as *JFace* offers implementation for a set of GUI concepts, which include a *text box*, *tree viewer*, and *context menu*. Framework-based applications are constructed by writing *completion code* that instantiates these concepts. The instantiation of such concepts requires various *implementation steps* in the completion code, such as subclassing framework-provided classes, implementing interfaces, and calling appropriate framework services.

For an application developer, the most important part of a framework is its *Application Programming Interface* (*API*). Unfortunately, the APIs of many modern object-oriented application frameworks are complex and difficult to use for two main reasons [61]. First, the sizes of the APIs are often large. For instance, the popular `Java Swing` framework has more than 800 classes and for instance, its `JTree` class by itself has about 400 methods [60]. Second, many of the existing frameworks suffer from the lack of proper manuals and documentation. To cope with these problems, application developers frequently follow the *Monkey See/-Monkey Do* rule [38] for framework-based application development: "Use existing framework applications as a guide to develop new applications."

While following the Monkey See/Monkey Do rule can be an effective application development strategy, understanding how an existing application implements a concept requires the ability to locate the code fragments implementing that concept. Unfortunately, locating the concept implementation can be challenging since these code fragments are often scattered in the application code and tangled with the implementation of other concepts. As an illustration of this problem, consider the task of implementing a *context menu* in a view of the *Eclipse* environment. Although the concept of context menu has a crisp manifestation at the graphical user interface, the implementation code typically involves multiple classes, such as *actions* and *menu managers*, and may be scattered across and tangled with code implementing other concepts. Consequently, despite the fact that a context menu is present in many existing Eclipse plug-ins, finding the relevant instructions among potentially many thousands lines of source code is a challenge.

Several categories of approaches have been proposed in literature to address this difficulty. *Framework usage comprehension* approaches such as *Strathcona* [57] and *FrUiT* [14] apply static analysis to the source code of example applications and allow retrieving code snippets or usage rules for a particular API element. These tools can be very helpful in understanding concept implementations, but they require the developer to know at least the names of some of the API elements involved in the concept implementation. However, they are less helpful if the developer has only a high-level idea of the concept that needs to be implemented or if the concept spans multiple classes or both. *Specification mining* approaches such as *CHRONICLER* [100] and *Perracotta* [142] use example applications to specify temporal and/or behavioral protocols that a program must follow when interacting with an API. Hence, they can not be particularly beneficial to the understanding of concept implementations on top of a given framework. Finally, *concept location* tools such as *SNIAFL* [149] and *SITIR* [80] could be used to locate parts of an application source code that implement a desired functionality or concept. However, these tools do not focus on framework API usage and, thus, the code identified using these tools will still include many application-specific instructions that are irrelevant from the viewpoint of framework usage.

This dissertation aims to address the above challenges in existing work and propose a framework comprehension technique that could be used by application developers to understand how to implement a concept of interest on top of a desired framework. The following section provides an overview of the proposed approach.

1.2 Approach Overview

Before providing an overview of the approach presented in this dissertation to address the problem of concept location for the purpose of framework comprehension, the nature of the problem is clarified using the context menu example discussed earlier in this chapter. For this purpose, consider the case of a developer creating a plug-in in the Eclipse environment. During development, the developer notices

that many plug-ins already present in the *workbench* implement a context menu. Interested in creating something similar, the developer has basically two choices. One is to search for help on documentation, mailing lists, and newsgroups. Another option is to look at the code of the plug-ins that implement the menu and search for the parts relevant to its implementation. However, as mentioned before, the implementation of such a simple framework-provided concept often requires that certain steps be employed in conjunction, not necessarily in a single code location. Moreover, such steps are frequently not well documented by framework developers and waiting for answers in mailing lists and newsgroups can become a frustrating, time-consuming task.

In order to tackle this problem, this dissertation introduces the notion of *concept implementation templates* and *FUDA* (*Framework API Understanding through Dynamic Analysis*), an approach to automatic extraction of such templates from traces of sample applications [55]. A concept implementation template is a Java-like pseudocode demonstrating the *implementation steps* that are necessary to instantiate a given concept. In particular, an implementation template specifies which framework packages to import, framework classes to subclass, framework interfaces to implement, and framework operations to call. Such a template can be used as a concise summary of the necessary implementation steps of a concept and a starting point to further investigate the concrete concept implementations in the sample applications. For the latter, FUDA templates provide traceability links between each implementation step in the template to its corresponding program statements in the sample applications.

In FUDA, a *framework-provided concept* is defined as a unit of functionality that is realized in the example application's source code by using the abstractions provided by the framework API. Furthermore, it should be possible to trigger the concept from the graphical or programmatic interface of the application. The key idea of FUDA is to take advantage of the information contained in sample applications to understand the implementation of a desired concept. To this aim, it utilizes dynamic analysis to determine the portions of the code in the sample applications that are relevant to the implementation of that concept. The goal is to be able to extract useful implementation templates using as few sample applications as possible to make the approach attractive in practise.

FUDA works as follows. Initially, the developer must identify a concept of interest, and select one or more sample applications implementing that concept in two or more different contexts. The next phase is to execute each sample application while a tracer tool collects a trace of calls that occur at the boundary between the application and the framework API. For the purpose of pinpointing the location of the concept execution in the trace, the developer informs the tracer tool the moments right before and after the invocation of the concept, a process called *marking*. In the subsequent *automated analysis* phase, the developer uses an analyzer tool to dissect the collected traces and extract the concept implementation templates. The automated analysis starts with the application of a novel slicing technique named *API trace slicing*. The goal of this step is to identify other events

that might be relevant to the implementation of the desired concept, and happen either before or after invoking it (*e.g.*, object creations as well as clean-up code), and hence, are not included in the marked events. Next, the sliced traces go through a process of removing application-specific content from the traces called *generalization*. Different kinds of facts about calls (*e.g.*, call nesting and object passing) are then extracted from these generalized traces. Afterwards, the extracted facts are intersected across all traces. The common facts in the intersection as well as the generalized traces provide the basis for further processing to generate a concept implementation template.

The tracer and the FUDA template extraction approach have been implemented as two separate but related Eclipse plug-ins. This prototype implementation was used in a study to evaluate the quality of the extraction process for twelve concepts with different characteristics on top of four widely-used frameworks. Some of these concepts were sampled from developers' forums. The study showed that the approach can produce templates with few false positives and false negatives for realistic concepts by using only two sample applications. Furthermore, a user experiment with twelve highly-skilled Java programmers was conducted to compare templates to framework documentation. For the studied sample, no statistically significant difference between using templates and documentation in terms of implementation time and number of introduced bugs could be detected. More specifically, for the studied sample, the choice of templates vs. documentation had much less impact on development time than the concept complexity.

However, the presented approach in this dissertation has some potential difficulties as well. First of all, it requires a number of executable sample applications that implement the desired concept. Finding these sample applications might be tough. Second, it should be possible to invoke the concept of interest from the graphical or programmatic interface of the application. This is not necessarily feasible for all kinds of concepts and frameworks. Finally, it requires the setup of the runtime environment which might not be easy in some situations.

1.3 Contributions

This dissertation makes the following contributions to the field of software engineering in general, and framework comprehension in particular:

- The API trace slicing as a novel slicing method that operates on traces representing the interactions between applications and a framework API rather than on complete instruction traces that is the case in traditional dynamic slicing approaches [2].
- The notion of automatically-extracted concept implementation templates that summarize the necessary implementation steps to realize a desired framework-provided concept on top of a target framework, such as which framework

packages to import, framework classes to subclass, framework interfaces to implement, and framework services to call.

- The FUDA framework comprehension technique as an approach to automatic extraction of concept implementation templates from API interaction traces.
- Prototype implementation of the FUDA framework comprehension technique as two separate but related Eclipse plug-ins for collecting API interaction traces and analyzing them to generate concept implementation templates.
- An empirical study with twelve realistic concepts, some of which from developer forums, with different characteristics on top of four widely-used frameworks that shows it is possible to extract the concept implementation templates with high precision and recall by using only two sample applications.
- A user study with twelve highly-skilled Java programmers in which the choice of templates vs. documentation had much less impact on development time than the concept complexity.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows:

- **Chapter 2** presents the necessary background information about object-oriented application frameworks.
- **Chapter 3** positions FUDA in the context of related work. For this purpose, the literature in the areas of framework documentation, framework usage comprehension, specification mining, concept location, aspect mining, and program slicing are surveyed and analyzed.
- **Chapter 4** explains the details of the FUDA framework comprehension technique.
- **Chapter 5** discusses the prototype implementation of FUDA and presents the details of an empirical study conducted to evaluate the precision and recall of the templates generated using the FUDA technique.
- **Chapter 6** provides a detailed description of a user study performed to evaluate whether FUDA templates can serve as a substitute for framework documentation when no documentation is available.
- **Chapter 7** presents discussion, concluding remarks, and future research directions.

- **Appendix A** presents the materials used during the template usage evaluation, such as different tutorials, various questionnaires, and specifications of concepts implemented by subjects.
- **Appendix B** provides the pseudocode of template generation algorithms.

Chapter 2

Object-Oriented Application Frameworks

This chapter provides the necessary background information about object-oriented application frameworks. In particular, Section 2.1 introduces the main properties of object-oriented application frameworks. Section 2.2 features how frameworks are used in the development of applications. Section 2.3 presents the advantages and disadvantages of frameworks from the viewpoint of framework usage. Finally, Section 2.4 summarizes this chapter.

2.1 Object-Oriented Application Frameworks

There are a number of definitions for object-oriented application frameworks. For example, a Taligent Inc. white paper [138] provides a definition that describes the *purpose* of a framework:"a framework is the skeleton of an application that can be customized by an application developer." Another definition by Roberts and Johnson [102] describes the *structure* of a framework:"a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact." Finally, a definition by Gamma et al. [39] covers both of these definitions:"a framework is a set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes, and defining their responsibilities and collaborations. A developer customizes a framework to a particular application by subclassing and composing instances of framework classes." With respect to these definitions, a framework captures the programming expertise required to solve problems in a specific domain; hides the parts of the design that are common to all applications in that domain; and makes explicit the pieces that need to be customized.

An important property of an object-oriented application framework that distinguishes it from traditional software libraries is that it often implements the main

control loop of the applications and dispatches events that the application-specific handlers respond to. This is called the *inversion of control* or the *Hollywood principle* [127] ("don't call us, we will call you") in computer programming. In other words, the framework itself is responsible for determining which set of application-specific methods to invoke in response to external events. Therefore, the application developer is told by the framework when and where his code will be called.

A software developer implements an actual application by customizing, configuring, and instantiating appropriate framework-provided classes and calling appropriate framework-provided services. Each application framework provides an *Application Programming Interface* (*API*) through which an application program can either specialize the framework code which is referred to as the *white-box reuse* or it can directly use the framework code which is referred to as the *black-box reuse*. The API of a framework constitutes a set of *hot-spots*. Hot-spots are those parts in the framework that are open to extension and customization. More specifically, hot-spots express those aspects of the framework domain that may vary in different applications of that domain and are expected to be specialized.

Fayad et al. [33] name four main characteristics for object-oriented application frameworks:

- *Modularity*. Frameworks improve modularity by encapsulating implementation details into stable interfaces. This will result in improved software quality since the effect of design and implementation changes is localized. This will also decrease the effort needed to comprehend and maintain existing software.

- *Reusability*. Reusability is resulted from framework's stable interfaces that provide generic building blocks to create new applications. Reusability allows to benefit from the domain knowledge of experienced developers who developed the frameworks. This can enhance the productivity of application developers and promote the quality, reliability, and interoperability of software.

- *Extensibility*. A framework defines explicit hook methods or hot-spots that allow applications to extend its stable interfaces. These hot-spots separate the stable interfaces that describe general behavior and interaction in the application domain from the variations required by different applications.

- *Inversion of Control*. This property was discussed earlier in this section. Unlike traditional software libraries, frameworks define the main control loop of applications, and hence, they may call the operations of the applications. Therefore, the application developer is responsible for implementing the operations (callback methods) that respond to notifications from the framework.

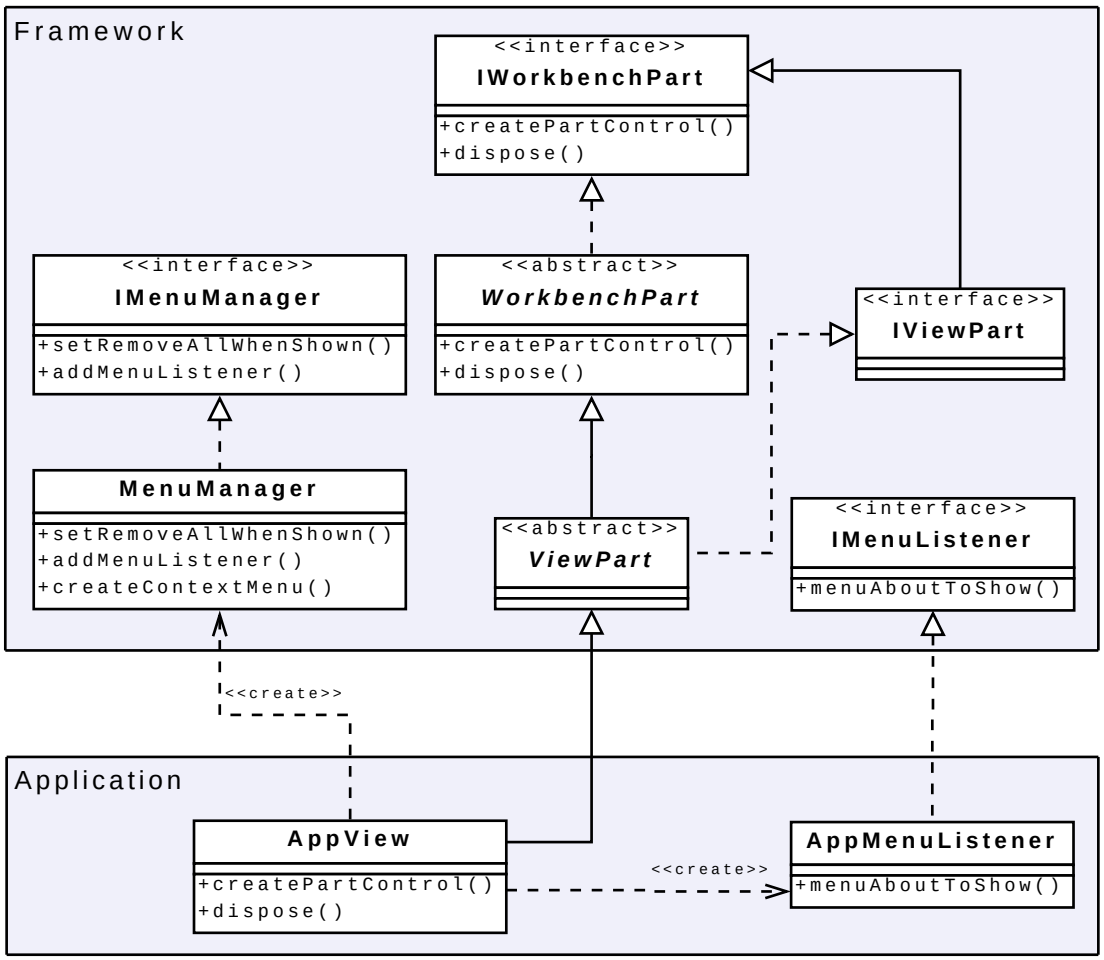


Figure 2.1: A view derived from the Eclipse framework

2.2 Framework Usage

Object-oriented application frameworks provide *domain-specific concepts* which are generic units of functionality. For example, a graphical user interface (GUI) framework such as JFace offers implementation for a set of GUI concepts, which include a *text box*, *tree viewer*, and *context menu*. Framework-based applications are constructed by writing *completion code* that instantiates these concepts through specializing framework hot-spots.

Depending on how a framework could be specialized, it is called a *white-box*, *grey-box*, or *black-box* framework [33]. The use of a white-box framework is based on inheritance and dynamic binding. A white-box framework is specialized by deriving new subclasses from the base abstract classes of the framework and by overriding and implementing required operations. On the other hand, in black-box reuse, applications are constructed through plugging together concrete classes (components) provided by the framework. Usually, a framework is not purely a white-box or a black-box framework. Instead, the framework specialization may

```
public class AppView extends ViewPart {
  TreeViewer viewer;
  ...
  public void createPartControl() {
    ...
    hookContextMenu();
    ...
  }
  private void hookContextMenu() {
    menuMgr = new MenuManager("#PopupMenu");
    menuMgr.setRemoveAllWhenShown(true);
    menuMgr.addMenuListener(new AppMenuListener());
    Menu menu = menuMgr.createContextMenu(viewer.getControl());
    ...
  }
  public void dispose() {
    ...
  }
  ...
}
public class AppMenuListener implements IMenuListener {
  public void menuAboutToShow() {
    ...
  }
}
```

Figure 2.2: Part of the code for creating an Eclipse tree viewer that contains a context menu on top of the Eclipse framework

utilize both white-box and black-box reuse which is referred to as grey-box reuse. In grey-box reuse, a part of the specialization is performed by composing existing components of the framework and the missing functionality is implemented through deriving new subclasses.

Figure 2.1 illustrates some parts of a class diagram for specializing the Eclipse framework for the purpose of creating an Eclipse view that contains a context menu. Figure 2.2 also displays its corresponding Java implementation and Figure 2.3 represents a screenshot of this view. This example is actually an example of grey-box reuse. The framework developer provides a set of classes and interfaces that can not be changed by application developers. Framework classes provide *service methods* that can be called by the application code and the rest of the framework methods are private implementation details. Examples of service methods in Figure 2.1 include `setRemoveAllWhenShown()`, `addMenuListener()`, and `createContextMenu()`. As Figure 2.1 illustrates, framework classes define the architectural skeleton to which the application program must conform. Application programs can define their own classes that may extend classes from the framework (*e.g.*, `AppView`) and may implement interfaces from the framework (*e.g.*, `AppMenuListener`). An application program may interact with a framework in ways that are permitted by object-oriented programming languages. These application-framework interactions include: subclassing (*e.g.*, `AppView`), implementing interfaces (*e.g.*, `AppMenuListener`), overriding superclass methods (*e.g.*, `dispose()`), calling framework service methods (`addMenuListener()`), creating instances of framework classes (*e.g.*, `MenuManager`), and holding on to framework objects (*e.g.*, `viewer.getControl()`). As could be

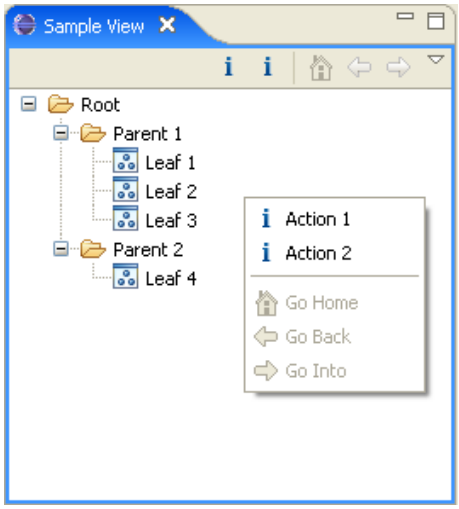


Figure 2.3: A screenshot of the Eclipse tree viewer resulted from the code in Figure 2.2

understood from this particular example, to implement a single functionality, application developers may require to employ multiple application-framework interactions in their programs.

At runtime, when the application program is started, the framework code starts running first and at selected times it will run the methods from the application program. These "selected times" are referred to as *callbacks* from the framework to the application code. Therefore, a callback is a transfer of control from the framework to the application program and is accomplished by the framework invoking a method defined in the application program, typically overridden framework methods. The application code does not run except during callbacks. Hence, callbacks are the only opportunity for the application program to affect the behavior of the framework. During callbacks, application programs can implement their own logic as well as calling the service methods on the framework. Callback methods often pass objects from the framework as parameters to the application program.

2.2.1 Differences between Application Frameworks and Software Libraries

Both application frameworks and software libraries offer the possibility to reuse existing code. However, when reusing software libraries, the application program always decides when to call the library. Libraries may require that requests from the application program have the appropriate kinds of parameters or that the requests are sequenced correctly. Nevertheless, libraries never require that the application program has a particular structure and they never call the application program. On the other hand, application frameworks call the application code, and it is this inversion of control that is unique to application frameworks as discussed earlier in this chapter.

A library publishes a list of provided services that it provides for application programs. Then, application programs are said to ask services that are defined by the library. On the other hand, in framework-based application development, the application program both provides and requires services of a framework. It provides services by implementing framework callback methods. It requires services by calling framework service methods.

2.3 Advantages and Disadvantages of Frameworks

Application frameworks have both advantages and disadvantages. These advantages make them attractive for creating new applications while their disadvantages hinder their usage. In the following, these advantages and disadvantages are discussed.

Hautamäki [48] names the following benefits for frameworks:

- *Reuse of both code and design.* As discussed earlier, one of the main benefits of frameworks is that they offer the reuse of both code and design. A framework is not simply a collection of classes, but also defines a generic design and helps the user to apply the underlying architecture.
- *Stored experience.* Frameworks are often implemented by developers who are expert in an application domain. Problems are solved once and design decisions are used again and again. In other words, by offering reusable design and code, frameworks reduce the amount of architectural decisions and implementation efforts that application developers have to make.
- *Improved software development cycle.* As a result of reusability of code and design that frameworks offer, application developers typically require to put less effort on programming, testing, and debugging.
- *Maintainability.* Applications developed on top of a framework utilize the same protocols and share the same architecture. This can result in better maintainability on top of mature frameworks. Nonetheless, this can be also a source of difficulty if the framework is not stable enough.

Despite several benefits that frameworks offer, they suffer from a number of difficulties as well. Fayad et al. [33] name the following difficulties with frameworks from the viewpoint of framework usage:

- *Framework development effort.* The advantages of developing frameworks are only obtained when they are reused several times. Nevertheless, it is hard to predict a framework's reusability beforehand. Especially, changes in the domain requirements can cause the framework to become outdated earlier than expected, and therefore, the framework development investment may not be cost-effective.

- *Learning curve.* One of the main problems in framework-based application development is the steep learning curve [33]. The complexity, variability, and abstract nature of frameworks' APIs make them difficult to learn. Moreover, since the classes in a framework are typically designed to work together, one may require to learn several classes all at once. Many modern frameworks also suffer from the lack of proper documentation which makes the situation even worse.

  To cope with these problems, many authors emphasize the role of example applications in learning frameworks [39, 33]. Example applications illustrate how the framework's classes can be instantiated, specialized, and used in an application to gain a desired functionality. Nevertheless, there are a number of difficulties associated with this approach as well. First of all, too much emphasize on learning frameworks from example applications can prevent application developers from going beyond the functionality that is explicitly present in the examples [126]. Locating the code that implements the desired functionality in the source code of example applications could be challenging as well since the code is often scattered across and tangled with code implementing other functionalities. Finally, finding the example applications themselves could be difficult too.

- *Integratability.* Software developers often require to use several frameworks altogether in the same application. However, integrating frameworks whose control loops are not designed to cooperate with other frameworks could be challenging.

- *Maintainability.* As the requirements of the framework change, the framework needs to evolve accordingly. Consequently, the applications developed on top of that framework require to evolve as well. Therefore, if the framework is not stable enough, its evolution can result in extra maintenance effort.

- *Validation and defect removal.* Validating and debugging applications implemented on top of frameworks could be challenging because of different reasons: (i) it is often difficult to distinguish bugs in the framework from bugs in the application code; (ii) the framework's inverted flow of control alternates between the application-specific method callbacks and application-independent framework infrastructure; and (iii) frameworks typically employ several levels of indirection (e.g., dynamic binding) to improve extensibility.

- *Lack of standards.* There are currently no standards for designing, implementing, documenting, and using frameworks.

2.4 Summary

This chapter presented the necessary background information about object-oriented application frameworks and differentiated them from traditional libraries. The

main property of frameworks that distinguishes them from traditional libraries is the property of inversion of control, *i.e.*, the framework decides when and where to call the client application code instead of the reverse.

For an application developer the most important part of a framework is its API. The framework's API consists of several variation points or hot-spots through which frameworks are specialized for specific applications. Although frameworks offer several advantages such as the possibility to reuse both code and design, they are often difficult to use for a number of reasons. Most importantly, they typically have the issues of complex APIs and incomplete user documentation. To cope with these challenges, application developers often benefit from example applications. Nevertheless, this approach could be problematic as well since the code implementing a desired functionality is often scattered in the source code of those applications.

Chapter 3

Related Work

This chapter presents an overview of related work in the areas of framework documentation (Section 3.1), framework usage comprehension (Section 3.2), specification mining (Section 3.3), concept location (Section 3.4), aspect mining (Section 3.5), and program slicing (Section 3.6).

3.1 Framework Documentation Approaches

As mentioned earlier, object-oriented application frameworks allow the reuse of both software design and code. Nevertheless, to be able to utilize frameworks effectively, appropriate framework documentation plays an important role [88]. Johnson [64] mentions three aspects that a framework documentation must cover: (1) the purpose of the framework, (2) how to use the framework, and (3) the design of the framework. However, we are interested in the usability of frameworks, and so, we concentrate on this aspect of framework documentation. The difficulty of the framework usability has led to the proposal of several framework documentation techniques, some of which are discussed below.

Cookbooks. Krasner and Pope [77] provide the API documentation in the form of *cookbooks*. Each entry in the cookbook is a *recipe* that explains a typical problem of using a framework and gives a stepwise guidance for solving it. Pree et al. later in [98] introduce the concept of *active cookbooks*. An active cookbook includes not only textual recipes, but also interactive elements to provide information on demand and to perform certain programming tasks in a semiautomated manner. However, documenting frameworks in the form of cookbooks is quite constraining. Cookbooks specify only a limited number of predefined ways of using a framework [20].

Ortigosa et al. [92, 91] introduce an advanced version of cookbooks called *SmartBooks*. SmartBooks supply instantiation rules describing the necessary tasks to be done in order to specialize the framework. These instantiation rules are used by an

automated intelligent agent to guide the application developer during the framework specialization process.

Froehlich et al. [35] document the framework hot-spots, *i.e.*, the way a framework is used, in the form of *hooks*. Hooks are similar in intent to textual cookbook recipes but are more formal and structured.

Patterns. Gamma et al. [39] define a *design pattern* as "a description of communicating objects and classes that are customized to solve a general design problem in a particular context". In other words, a design pattern describes a recurring solution to a common design problem. Gamma et al. [39] also introduce a catalogue of twenty-three design patterns as an attempt to help software engineers reuse successful designs. Design patterns can be used to document the framework's design as well as the way it is commonly used. The structure of a pattern is often represented as a class diagram which works as a template outlining the roles and the relationships for the program elements (*e.g.*, classes, methods, or fields) participating in the solution [125].

Johnson [64] was the first to propose documenting frameworks using patterns. For this purpose, he introduced an informal pattern language in which each pattern describes a problem that occurs commonly in the problem domain of the framework, and also describes how to solve that problem. Later, he used design patterns to document the frameworks [65].

Pree [97] introduced the notion of *meta-patterns* as abstractions of design patterns that describe how to construct frameworks independent of a specific domain. The notion of meta-patterns was later used by Tourwé and Mens [122, 120] to document framework hot-spots and support framework-based software evolution.

Hakala et al. in *FRamework EDitor* (*Fred*) project [45] use *specialization patterns* to define the specialization interface (*i.e.*, hot-spots) of the framework. Fred supports both the framework developers in creating the specialization patterns for the framework, and the application developers in specializing the framework by providing a step-by-step task list implied by the specialization pattern. Fred dynamically adjusts this task list according to the choices made by the programmer, and checks that the constraints of the framework are not violated.

Recently, Fairbanks et al. [32] introduced *design fragments* as a pattern that encodes a conventional solution to how a program interacts with a framework to accomplish a goal. The application developer has to manually browse the catalog of available design fragments and select the one that fits her requirement. Then, she has to manually bind the selected design fragment to the application code. Afterwards, the conformance of the application code to the selected design fragment is automatically checked and violations are reported.

Specific Approaches. Fontoura et al. [34] propose *UML-F*, a special profile of the UML for representing frameworks' API. The UML-F provides special UML

stereotypes and tags for annotating UML diagrams to encode framework constraints. Application developers can use a UML-F model to create the design of their framework-based applications. An automated wizard then transforms the model into code.

Hou et al. [59] use the *Framework Constraint Language* (*FCL*) to specify the interface between the framework API and the user code such that the specification describes all legal uses of the framework API. The language has a number of built-in predicates. The semantics of FCL is based on a first-order logic extended with set and sequence operations.

Antkiewicz and Czarnecki [7, 6] propose to use *Framework Specific Modeling Languages* (*FSML*s) to model how framework-provided abstractions are used in the framework completion code, and automatically check for violations of framework rules. The abstract syntax of an FSML defines a decomposition of a framework-provided concept into a hierarchy of mandatory and optional *features* represented as a *feature model* [22]. Features represent distinguishing characteristics of concepts and can be used to discriminate among concept instances. Consequently, concept instances are described by configurations of features.

Although framework documentation approaches can help application developers learn how to implement a concept of interest on top of a framework, software frameworks often lack complete documentation, and not all the framework-provided concepts are necessarily documented by the framework developers. Moreover, framework documentation requires manual effort and consequently, documentation of the framework may become outdated as the framework evolves. On the other hand, the FUDA framework comprehension technique presented in this dissertation can automatically extract implementation templates for concepts of interest from example applications implementing those concepts. As will be described later in Chapter 6, a user study with twelve highly-skilled Java programmers shows that FUDA templates can serve as a substitute for framework documentation when no documentation is available. In particular, FUDA templates can be regenerated automatically for new releases of a framework.

3.2 Framework Usage Comprehension

Although framework documentation approaches can help framework comprehension, their main difficulty is that manual effort is needed to create the corresponding documentation. Nonetheless, framework developers are not often interested in creating suitable documentation. Therefore, many modern application frameworks have the difficulty of incomplete user documentation. On the other hand, object-oriented application frameworks are often accompanied by several example applications showing how to specialize those frameworks for different purposes. Consequently, several framework comprehension approaches have been proposed in literature that benefit from example applications for the purpose of framework

usage comprehension. These approaches often recommend the application developers to follow the *Monkey See/Monkey Do* rule for framework-based application development: "Use existing framework examples as a guide to develop new applications" [38]. This section provides an overview of these techniques in three different categories: code assistants, framework API comprehension, and example application finders.

Code Assistants. There are a number of approaches that use existing framework applications as a guide to aid application developers during programming. Examples of these approaches include *Prospector* [85], *PARSEWeb* [116], *XSnippet* [106], *Strathcona* [57], *CodeWeb* [89], and *FrUiT* [14]. Given two API types τ_{in} and τ_{out} as a query, both Prospector and PARSEWeb mine for code snippets that transform an object of type τ_{in} to another object of type τ_{out}. For this purpose, Prospector mines *signature graphs* generated from API specifications and *jungloid graphs* representing code, and ranks the results by the shortest path from τ_{in} to τ_{out}. On the other hand, PARSEWeb interacts with the Google code search engine[1] to gather relevant code samples. Then, it extracts method call sequences from the directed acyclic graphs (DAGs) built out of the abstract syntax trees (ASTs) of the collected code samples.

Both XSnippet and Strathcona are context-sensitive code assistant tools that maintain a repository of code snippets. During application development, they compare the context of the programming task at hand with the code snippets in the repository and recommend relevant coding examples. In XSnippet, relevancy is defined by the context of the code, both in terms of the parents of the class under development as well as lexically visible types. On the other hand, Strathcona applies six heuristics separately that match the structural context descriptions encapsulated in the developer code with that encapsulated in the example code. The result is a set of code snippets that occur most frequently when applying all proposed heuristics.

Both CodeWeb and FrUiT mine for frequent API usage patterns as *association rules*, *e.g.*, *subclass* $A \Rightarrow$ *call* m, and keep them in a repository. FrUiT uses such rules to automatically suggest implementation steps that are relevant in the context of the current programming task.

All the approaches mentioned here are mainly code assistants in the context of a programming task at hand and thus are useful to the understanding of fine-grained framework extensions, such as how to call a specific framework method or how to instantiate a particular framework class. Nevertheless, in contrast to this dissertation, they do not provide a complete code snippet or implementation template for instantiating a coarse-grained concept which may span multiple framework classes and methods. Moreover, these approaches are most helpful when the developer knows at least the API elements of interest; they are less helpful if the developer has only a high level idea of the concept that needs to be implemented.

[1] `http://www.google.com/codesearch`.

In contrast, FUDA uses primarily dynamic analysis to allow developers to identify their concepts of interest by invoking them explicitly, while all these approaches use static analysis and hence do not support concept identification by invoking concepts directly from the application user interface. Consequently, in FUDA, as will be described later in Chapter 4, the developer does not need any knowledge of the framework API other than the names of the packages in which it resides.

The advantage of static analysis is that it can cope with large body of applications and potentially incomplete code. However, since FUDA applies dynamic analysis, it requires complete executable code of applications and needs setting up the applications' runtime environment. Nonetheless, the advantage of dynamic analysis is that it can handle highly polymorphic and reflective code, which is often a part of modern frameworks.

Framework API Comprehension. Viljamaa in *Pattern Extractor* [125, 126], Thummalapenta and Xie in *SpotWeb* [117], and Schäfer et al. in [110] propose approaches in which sample applications of the framework are used to provide a better understanding of the framework API. The main aim of these approaches is to provide a starting point for the developers to investigate and learn the framework API. In Pattern Extractor, the formal concept analysis [40] is adapted to reverse engineer the framework's API and produce *Fred specialization patterns* (see Section 3.1) from the Java source code of the framework itself and its example applications. SpotWeb mines the framework's sample applications gathered from open-source repositories to determine the framework's *hot-spots* and *cold-spots*. Hot-spots are defined as API classes and methods that are frequently used, while cold-spots are those that are rarely used in the sample applications. Finally, in Schäfer et al.'s approach, a clustering technique that employs usage data from framework instantiations is applied to group the elements of a framework API into building blocks that are typically used together. The assumption underlying this approach is that the more often framework elements are used together by existing sample applications, the more likely these elements are conceptually related to each other. This implies that they should be considered together during framework-based application development.

Similar to FUDA, these approaches also aim to provide a starting point for developers to investigate and learn the framework API. However, in contrast to FUDA, which provides an implementation template for a concept of interest, these approaches aim to provide a general view of the framework API and they do not target any specific framework-provided concept. In particular, they do not provide any example code snippets for implementing a desired framework-provided concept that is the goal of this dissertation.

Example Application Finders. All the approaches introduced so far in this section benefit from sample applications for the purpose of framework comprehension. Therefore, finding appropriate example applications plays an important role in these framework comprehension approaches. To address this requirement,

a number of techniques, such as *Assieme* [56] and *XFinder* [24], are proposed in literature that aim to find example applications for a desired framework.

Assieme is a special purpose Web search engine that allows developers to ask queries attempting to identify an appropriate API for a problem, seeking more information about a particular API, and seeking samples that use an API. For this purpose, Assieme combines information from Web-accessible Java Archive (JAR) files, API documentation, and Web pages that include explanatory text and sample code. Even though it has been shown that Assieme can generate better results than general-purpose Web search engines such as Google, Assieme may still generate many results and it is up to the developer to discriminate or to choose something among the results. Moreover, the example applications found by Assieme may contain lots of application-specific code and it is up to the user to understand which parts of the code implement her target concept. On the other hand, in this dissertation, the developer has already a number of example applications at her hand and she is interested in seeing how those sample applications implement her desired concept.

XFinder is an extension of *Mismar* [23], a concept-oriented documentation toolset that focuses on code artifacts and their relationships. XFinder accepts concept implementation templates expressed in Mismar as input and tries to find instances of those templates in its code base. Therefore, XFinder does the reverse of the work done in this dissertation. More specifically, FUDA uses example applications to generate concept implementation templates, while XFinder uses templates to find relevant example applications.

3.3 Specification Mining

Although generating specifications has been studied for decades in different areas of software engineering [132, 31, 81, 19], the term *specification mining* was first introduced by Ammons et al. [5] as an approach for automatically discovering the protocols (or rules) that a program must follow when interacting with an API (or library). The focus of this section is on this kind of specifications. A large number of work on this topic have been reported in literature which can be mainly classified into *static* [1, 78, 137, 79, 114, 101, 50] and *dynamic* [5, 83, 142, 84, 82, 109] approaches. Static approaches extract the specifications directly from the source code while dynamic approaches infer the specifications from the runtime traces. In the following paragraphs, an overview of these approaches is provided.

Static Approaches. Interacting with APIs (or libraries) requires following a protocol such as following a specific set of method calls. Many of the static specification miners analyze the source code of client applications to generate specifications of method calls [141, 1, 79]. Xie and Pei [141] introduce *MAPO*. Given a query that characterizes an API by a method, class, or package, MAPO searches relevant

source files from open source repositories available on the Internet. Then, method call sequences are extracted from those source files. Afterwards, sequential patterns are mined using the closed sequential mining technique, and the results are presented to the user at the end. Acharya et al. [1] present a methodology to automatically extract API usage scenarios as partial orders among user-specified APIs, directly from API client source code. For this purpose, they statically generate traces related to the APIs of interest. Next, they apply a frequent partial order miner named *FRECPO* [94] to generate the specifications. LtRules implemented by Liu et al. [79] extracts temporal specifications from known good programs using the *BLAST* [51] model checker. More specifically, it uses a number of predefined templates to generate all candidate temporal specifications, and checks the candidate specifications against known good client programs using the BLAST model checker. Those candidate specifications that pass the BLAST test are considered to be specifications.

Some other work on static specification mining accept a type as input and generate a temporal specification in the form of an automaton that encodes legal call sequences of operations on that type. A call sequence is considered legal if it does not lead to an assertion failure or exception. For instance, *Static WML* [137] infers pairs of methods in a class that may raise an exception when called successively. Other examples include *JIST* [4] presented by Alur et al. and *Permissive Interfaces* [50] introduced by Henzinger et al.

A number of approaches (*e.g.*, [78, 131, 100]) not only determine protocols of method calls, but also detect violations of those protocols in the source code of client applications. For instance, Li and Zhou developed *PR-Miner* [78] in which they use frequent itemset mining to extract highly correlated function calls from the source code of a software system. Next, they use the obtained correlated function calls as specifications and analyze the source code to find instances of sets of function calls that violate those specifications to detect bugs. A similar approach named *JADET* [131] is proposed by Wasylkowski et al. in which the tool uses frequent closed itemset mining technique to infer ordering patterns among method calls and to detect abnormal usages. Ramanathan et al. in *CHRONICLER* [100, 101] infer *function precedence protocols* in the form of "a call to procedure `q` must always be preceded by a call to procedure `p`". For this purpose, their approach collects predicates along each distinct path to each procedure call and analyzes them using sequence mining techniques to create a collection of feasible precedence protocols. Deviations from these protocols found in the program are tagged as violations, and represent potential sources of bugs.

Dynamic Approaches. The work by Ammons et al. [5] is the pioneering work on dynamic specification mining in which they introduce a machine learning approach to mine specifications that reflect temporal and data dependency relations of a program through traces of its API-client interaction. The specifications discovered model the API-client interaction protocols and are expressed using a *finite*

state automata (*FSA*). To handle traces of buggy applications, their approach requires human experts to decide whether a violation is really a bug. Lo and Khoo in *SMArTIC* [83] aim to improve the specifications mined by Ammons et al. by (1) early identifying and filtering the erroneous traces from the input traces, and (2) clustering related traces to localize the learning process to groups of related execution traces. Moreover, they use *probabilistic FSA* (*PFSA*) instead of FSA to express the specifications mined.

Perracotta [142] developed by Yang et al. generates temporal API rules involving only two events, *i.e.*, of the form of $A \rightarrow B$, from dynamic traces. For instance, acquiring a lock should be eventually followed by releasing the lock. In order to generate longer rules, they propose the notion of *chaining*. For example, if both $A \rightarrow B$ and $B \rightarrow C$ are significant, they can be merged to form $A \rightarrow B \rightarrow C$ which will be significant as well. Lo et al. in *CLIPER* [84] also mine for temporal patterns, but in the form of *iterative patterns*, defined as the "commonly occurring series of events exhibited repeatedly within a sequence [trace] and across multiple sequences [traces]". Using CLIPER, it is possible to capture temporal patterns that may occur multiple times within traces because of loops and may include more than two events, while using the Perracotta it is impossible to do so. Some other work that extract temporal specifications from dynamic traces include *Quark* [82], *ADABU* [25], *Javert* [36], *GS* [37], and *WN* [133].

Sankaranarayanan et al. [109] propose a technique to automatically infer declarative specifications of the API behavior for target concepts such as the raising of an exception, the return of a value by a function, or the printing of a specific message onto the output. To this aim, their approach runs several unit tests on the library for the target concept. Next, the data collected by these runs are fed into an inductive learner to obtain specifications expressed in datalog/Prolog.

Different from framework usage comprehension techniques presented in Section 3.2, the specification mining approaches mainly extract behavioral and temporal protocols for interactions between the applications and the APIs of libraries, while framework usage comprehension approaches provide aid for understanding the framework API and/or framework-based application development. In contrast to specification mining approaches, FUDA also does not recover API interaction protocols. The latter is important for library API usage, but less so for frameworks. Frameworks typically follow *inversion of control* by enforcing protocols in framework rather than application code.

Additionally, dynamic specification mining techniques often require several runtime traces in order to recover different legal execution sequences. On the other hand, as will be discussed in Chapter 5, in this dissertation we aim to keep the number of traces as small as two to make the FUDA technique attractive in practise.

3.4 Concept Location

Identifying the relevant parts of an application source code that implement a desired functionality or requirement is an important problem in program comprehension and maintenance; it is often referred to as the *concept* or *feature location* problem in software engineering literature. Although it might be easily possible to manually perform the concept location in small software systems, for large and complex systems, it could be impractical without tool support. To address this issue, a number of techniques have been proposed in literature [104, 44, 149, 112, 9, 80, 29]. Depending on how the information is extracted from the source code, these techniques can be classified into three main categories: *static* [104, 44, 17, 112, 149, 86], *dynamic* [9, 139, 30, 140], and *hybrid approaches* [29, 80]. Static approaches are based only on the information statically retrieved from the source code without actually executing it. On the other hand, dynamic approaches collect information during runtime by using a number of use cases. Hybrid approaches combine the two.

The difficulty of the pure dynamic approaches is that they are highly dependent on the input data that are considered during the runtime to collect information and generalizing from this data might not be safe. Additionally, they often require several use cases and designing these use cases might be hard, especially when the concept is not user-triggerable. On the other hand, static approaches are more conservative and safer. However, since many interesting properties of programs are statically undecidable (*e.g.*, which paths are taken at runtime), static analyses are limited to approximative solutions that might be practically imprecise. Hybrid approaches try to solve these problems and take the advantages of both static and dynamic techniques. In the rest of this section, an overview of different concept location techniques is provided.

Static Approaches. As mentioned earlier, static concept location techniques extract information regarding the implementation of a given concept directly from the subject application source code without executing it. There are three main categories of static concept location techniques: exploratory approaches, lexical code searchers, and information retrieval (IR)-based techniques.

Exploratory Approaches. Exploratory approaches typically provide tools by which users can interactively explore and/or query the source code of an application. These techniques are mainly based on the assumption that the user has some initial knowledge about the implementation of the concept at the beginning and then builds up her knowledge during the process of concept location [148, 149]. Consequently, these approaches are iterative, require a lot of human involvement, and are hard to automate. Examples of these approaches are *FEAT* [104, 103], *ConcernMapper* [105], *Active Models* [21], *JQuery* [63], and *Sextant* [28, 111].

Chen and Rajlich [17] propose the idea of a computer-assisted concept location

process using the *abstract system dependence graph* (*ASDG*). An ASDG illustrates the dependencies among routines, types, and variables at an abstract level. The navigation on the ASDG is computer-assisted and the user takes on the search for a concept's implementation.

Robillard et al. present *FEAT* [104, 103] and *ConcernMapper* [105] as Eclipse plug-ins that allow a software developer to iteratively create, visualize, and analyze *concern graphs*, where concepts are referred to as concerns. A concern graph localizes an abstracted representation of a concern by explicitly documenting the dependencies among the different sections of source code that play a role in implementing that concern. This abstracted representation of the concern can be mapped to the program source code to recover further implementation details.

Coelho and Murphy introduce an *active model* [21] as a structure diagram that aims to reduce graphical complexity by presenting the right information about the crosscutting concern to the developer at the right time. For this purpose, an active model initially uses a description of the crosscutting concern generated by other tools (*e.g.*, FEAT). Next, by using a combination of interaction techniques and a few automated operations, it specifies further crosscutting concern information of interest and displays them to the user. The user continues interacting with the model through the provided automated operations until she is satisfied with the investigations. In this way, the developer can focus on a slice of the model of the entire system.

Janzen and De Volder devise *JQuery* [63], a code query tool which is implemented as an Eclipse plug-in on top of a modified version of the TyRuBa [128] logic programming language as its query engine. The syntax of the TyRuBa is very similar to that of Prolog. JQuery provides a tree-viewer browser for displaying and navigating query results. In JQuery, a software developer can define her own top-level logic queries and execute them against a logic database representation of her source code. Nodes in the tree can then be queried individually in the same manner, allowing further investigation of the complex web of relationships that exist among scattered elements of code. In this way, investigations consisting of multiple searches can be done within a single window without losing the context.

Eichberg et al. introduce *Sextant* [28, 111], a graph-based software exploration tool that follows a query-based approach for both searching program elements and browsing along different kinds of relations among the program elements. Nodes in Sextant graphs represent program elements discovered in the search activity and edges represent relationships among those program elements. In Sextant, it is possible to navigate through software systems that include several kinds of artifacts. For example, it is possible to navigate from the source code to some configuration files or XML deployment descriptors and back. Similar to JQuery, all navigations are visualized in Sextant in one view and hence, all single steps of the exploration process are kept connected.

Lexical Code Searchers. Some approaches provide facilities to lexically search the

source code of applications using regular expression queries to locate concepts. *AspectBrowser* [43], *Prism* [145], and *Find-Concept* [112] are examples of this kind of approaches. AspectBrowser allows users to enter `grep`-like regular expressions. Then, it lexically searches programs for those expressions and visually represents the results. Prism also searches the source code for lexical patterns. However, it provides an automated advisor that provides suggestions for improving the accuracy of the queries. Finally, Find-Concept expands search queries with synonyms to locate concerns more accurately in code and also provides source code exploration facilities to help users explore the code based on the results of the queries.

The problem of lexical code searchers is that they are based on the assumption that strict naming conventions are followed in the source code. Moreover, the quality of the result is highly dependent on the regular expression queries and the results may include lots of false positives and false negatives. For instance, features like morphology changes, line breaks, and reordered terms may cause regular expression queries to fail [112].

IR-based Approaches. Information retrieval (IR) is the process of determining the relevant documents from a collection of documents, based on a query presented by the user (*e.g.*, search strings in Web search engines). There are a number of concept location techniques (*e.g.*, [149, 86]) that assume the identifiers and the comments in the code contain a significant amount of useful information and apply IR techniques to identify relevant parts of a source code that implement the desired concept.

Zhao et al. [149] introduce *SNIAFL* as a *Static, Non-Interactive Approach for Feature Location*. In SNIAFL, users describe their features of interest using natural languages. Then, an information retrieval (IR) technique is used to identify the basic connections between features and source code computational units (functions in this approach). However, because of the imprecision in the results of IR techniques, they also use a static representation of the source code named *BRCG* (*Branch Reserving Call Graph*) [99] to further retrieve computational units that may play a role in implementing the desired concept.

In another approach presented by Marcus et al. [86], the concepts expressed in natural languages are treated as queries and the *LSI* (*Latent Semantic Indexing*) IR technique is applied to retrieve the relevant parts of the source code. LSI derives the meanings of terms from their usage in the text, instead of a predefined dictionary, which is an advantage over the lexical code searchers. Unlike SNIAFL which uses BRCGs to automatically investigate the results further, in Marcus et al.'s approach, the results should be processed further manually by the user herself.

The same as the lexical code searchers, these approaches also suffer from the assumption that the developers have followed a specified naming convention. Nevertheless, IR-based approaches are less sensitive to poor search queries compared to lexical code searchers.

Dynamic Approaches. Dynamic concept location techniques require a number of use cases that exercise the target concept at runtime, and may require some that do not. The collected traces are then analyzed and mapped to the source code. One difficulty of dynamic approaches is that execution traces are usually large and contain lots of irrelevant events. Consequently, as described below, these approaches typically require at least two traces such that the role of one is to remove noises from the other one [80].

Wilde and Scully [139, 115] introduce the *Software Reconnaissance* technique in which the program is executed using some test cases "with" the desired concept and some test cases "without" the desired concept. After that, the set of program elements executed in the "without" tests are subtracted from the set of elements executed in the "with" tests and the "marker" elements for the desired concept are found. A similar approach is reported by Wong et al. [140], where the main difference is that it can help to identify code that is unique to a concept or common to a group of concepts at different levels of granularity (*i.e.*, files, functions, blocks, lines of code, etc.).

Dynamic Feature Traces (*DFT*) is proposed by Eisenberg and De Volder [30]. Similar to two techniques above, the DFT approach is also based on analyzing the execution traces collected by running the system using a set of test cases that invoke the target concept and a set of test cases that exclude it. However, the main difference between DFT and the other two techniques is that it uses some heuristics to rank the relevance of the code elements found to a concept.

Antoniol and Guehénéuc [8, 9] extend the Software Reconnaissance technique by introducing the *Scenario-based Probabilistic Ranking* (*SPR*) approach with the aim of concept location in large, multi-threaded, object-oriented software systems. For this purpose, they use processor emulation, knowledge-based filtering, and probabilistic ranking of events happened in the marked traces.

Salah and Mancoridis [107, 108] identify concepts in terms of marked traces, *i.e.*, they demarcate the beginning and the end of a concept's execution at runtime using a trace marker utility. These marked traces are also used to provide a hierarchy of views at different levels of abstraction capturing the concept interactions.

Hybrid Approaches. As discussed before, dynamic concept location techniques typically require more than one execution scenario to filter noise from dynamic traces. Hybrid approaches try to address the same issue by using static information. It has been shown that hybrid approaches are typically more effective than pure dynamic or pure static ones in terms of the quality of the results [95].

The pioneering work of hybrid concept location is the work done by Eisenbarth et al. [29]. In this work, the execution traces collected at runtime by invoking the concept or a set of concepts of interest are represented in the form of a *concept lattice* [40]. Then, based on this lattice, various relationships between features and computational units can be easily recovered. To further refine the results achieved

by this concept analysis, a manual static analysis based on the dependency graph is applied next. Zhao et al. [147], however, mention that the scalability, complexity, and visualization of concept lattices are the major issues of this approach when it is applied to the exploration of a large program.

Poshyvanyk et al. in *PROMESIR* [95, 96] combine the static IR-based technique that uses LSI [86] with the dynamic approach based on the probabilistic ranking of events SPR [8, 9]. Both these approaches were introduced earlier in this section. Developers use the LSI-based approach to query the source code and to get a ranked list of facts probable to be relevant to the given concept. Using SPR, the user can analyze dynamic traces of execution scenarios and obtain a ranked list of elements (*e.g.*, methods and classes) likely to be relevant to the concept exercised under the given scenarios. Then, these two ranked lists are combined using an *affine transformation* [62].

Liu et al. propose *SITIR* [80], a hybrid concept location technique which is based on the assumption that a single execution trace, collected by invoking the concept of interest, includes all the necessary information to locate the main parts of the source code that are implementing that concept. With respect to this assumption, they collect an execution trace by invoking the desired concept at runtime and apply the LSI information-retrieval technique to filter noise from the collected trace based on a user-defined query expressed in natural languages. To reduce the size of the trace, they also use trace marking at runtime. Finally, the results are inspected by the programmer.

None of the concept location techniques discussed in this section explicitly address the issue of concept location in the context of framework comprehension. In other words, unlike the FUDA framework comprehension technique, all of these concept location techniques focus on retrieving concepts in general application code rather than framework-provided concepts. Consequently, the results may contain many application-specific instructions that are irrelevant from the viewpoint of framework usage. FUDA avoids this problem by focusing on API interaction traces and removing the application-specific content from those traces.

While FUDA enables the use of different sample applications implementing a desired concept, all the dynamic and hybrid concept location techniques discussed here except *SITIR* [80] work with different traces from a single application. The use of different applications affords more diversity in the traces and can lead to better results with fewer traces. In some cases, using different applications may even be the only way to obtain any useful results. For example, obtaining traces that use an Eclipse tree viewer in different ways from one application may not be possible.

Finally, unlike FUDA, none of the dynamic or hybrid concept location techniques discussed here uses API trace slicing with API trace marking. In particular, although the SITIR [80] and the approach by Salah and Mancoridis [107, 108] for concept location use runtime trace marking to reduce the size of the traces, none of them apply slicing or any other technique to identify relevant events to the im-

plementation of the desired concept that happen before or after the marked events and they will lose them. However, FUDA is able to identify such relevant events by applying a novel API trace slicing approach.

3.5 Aspect Mining

In *aspect-oriented programming* (*AOP*) [69], aspects modularize the design and development concerns that cut across parts or all of the traditionally developed software systems. One of the important activities in applying AOP onto legacy software systems is *aspect mining* [113]. Compared to concept location, aspect mining can be seen as the reverse activity. More specifically, in concept location, developers explicitly search the software systems for known concepts or crosscutting concerns of interest. On the other hand, in aspect mining, developers search the source code to identify unknown crosscutting program concerns in software systems and modularize them into aspects. The same as the concept location techniques, the aspect mining approaches can be also mainly classified into *static* [87, 15, 121, 47, 13], *dynamic* [12, 119], and *hybrid* [11] ones. Some of these approaches are briefly discussed below.

Marin et al. [87] devise an aspect mining technique based on *fan-in analysis*. The idea is that if a method is called many times from many different places, it is likely to represent a functionality that is scattered across the code. Based on this idea, the set of methods that have a fan-in above a certain threshold are analyzed (largely manually) and candidate aspects are identified.

Bruntink et al. [15] use the output of code clone detectors (*e.g.*, *CCFinder* [66]) to identify aspect candidates. A code clone is defined as a set of code fragments that are duplicated (or cloned). In this work, a grade is attached to each code clone based on some metrics. Code clones that are scored above a threshold value are then chosen as the aspect candidates.

Tourwé and Mens [121] introduce an approach for aspect mining based on analyzing the identifiers used in the program. The assumption in their approach is that programmers often follow naming conventions to associate related but distant program entities. They use formal concept analysis to group program entities with similar identifiers. Then, the groups which have at least a certain number of elements (*e.g.*, 4) are marked as seeds for potential aspects. These seeds are next manually analyzed by the user to identify valid aspects.

Tonella and Ceccato [119] suggest a pure dynamic approach in which they apply formal concept analysis to identify the relationship between execution traces and executed computational units. In another work, Breu and Krinke propose a method named *DynAMiT* [12] in which program traces reflecting the run-time behavior of a system are investigated for recurring execution patterns. Breu also reports on a hybrid approach in [11] where the dynamic information of the previous DynAMiT technique is complemented with static information such as static object types.

As with aspect mining approaches, the FUDA framework comprehension technique also addresses the problem of understanding crosscutting concerns. Nevertheless, FUDA aims to locate the implementation of user-specified framework-provided concepts that may be scattered across and tangled with code implementing other concepts, while aspect mining methodologies search for unknown crosscutting concerns and aim to modularize them into aspects. Additionally, FUDA does not extract the scattered code segments into separate aspect modules, but leaves them inline and supports their localization through traceability links between the concept implementation templates and sample applications.

3.6 Program Slicing

Program slicing was originally defined by Mark Weiser [134, 135] as an executable subset of program statements that preserves the original behavior of the program with respect to a subset of variables of interest `V` at a program point `p`. In other words, the values of the variables in `V` at program point `p` are the same in both the original program and the slice. Weiser's slices are executable programs that are constructed by removing zero or more statements from the original program. After original definition by Weiser, several variations of program slicing with various applications in software engineering have been introduced in literature. Some of these variants are *static slicing* [135], *quasistatic slicing* [124], *dynamic slicing* [73], *simultaneous dynamic slicing* [46], and *conditioned slicing* [16]. In the following, we talk in more detail about static slicing and dynamic slicing which are the main categories of slicing techniques.

Static Slicing. The original definition of program slicing by Weiser is referred to as static slicing. However, Weiser himself in [136] showed that computing the minimal subset of program statements that satisfies his definition of slice is undecidable and approximation algorithms are required. For the purpose of this approximation, he used data flow analysis on control flow graphs to compute *intraprocedural* and *interprocedural* slices. Intraprocedural slicing refers to slicing a program that consists of just one method with no calls, while interprocedural slicing refers to slicing a program consisting of different methods and calls.

Ottenstein and Ottenstein [93] provided a more efficient algorithm for intraprocedural slicing by defining a slice as all program statements and predicates that may influence the value of a variable `v` at a program point `p`. This definition of slice has been also referred to as *backward slice* in contrast to *forward slice* which is defined as all program statements and predicates that may be influenced by the value of variable `v` at program point `p`. Ottenstein and Ottenstein used a graph reachability algorithm on a *Program Dependence Graph* (*PDG*) to compute slices. A PDG consists of nodes representing the statements of the program, while edges carry information about control and data dependencies among them. In this algorithm,

backward (forward) intraprocedural slicing from program point `p` is backward (forward) reachability in the PDG from node `p`.

Horwitz et al. [58] extended the PDG based algorithm to an interprocedural version that uses the *System Dependence Graph* (*SDG*) to compute backward (forward) slices. A SDG consists of a collection of PDGs for each procedure and interprocedural edges connecting calls to entries, actual parameters to formal parameters, and procedure results to call sites. Although the Horwitz's algorithm was originally designed for procedural languages like C, the same ideas work for a subset of Java that excludes threads and exceptions. Later work by Ball/Horwitz [10], Choi/Ferrate [18], and Allen/Horwitz [3] extended the algorithm to handle jumps (*e.g.*, `goto`, `break`, etc.) and exceptions (*i.e.*, `try`, `catch`, `throw`).

Dynamic Slicing. Static slicing does not make any assumptions regarding the program inputs. In other words, static slicing preserves the value of a variable `v` at a program point `p` for all program inputs, and therefore, it covers all program executions. However, there can be situations where one is interested in a slice that preserves the program's behavior for a specific program input. This type of slicing is called *dynamic slicing* and was originally introduced by Korel and Lasky in [72]. In other words, the purpose of dynamic slicing is to identify all the program statements that affect a variable of interest in a particular program execution.

There are two main types of dynamic slicing approaches [76]: *executable dynamic slicing* and *non-executable dynamic slicing*. The aim of executable dynamic slicing approaches is to produce dynamic slices that are executable. For example, the original dynamic slicing approach presented in [72] provides an iterative algorithm based on data flow and control flow dependencies to compute executable dynamic slices. In this algorithm, if any occurrence of a statement in the execution trace is included in the slice, then all other occurrences of that statement are also included in the slice.

Non-executable dynamic slicing approaches produce slices that are not necessarily executable. For example, Agrawal and Horgan [2] introduced an approach for computing dynamic slices that uses a *Dynamic Dependence Graph* (*DDG*) to take into account different occurrences of a given statement that might be affected by various sets of statements due to redefinitions of variables. In the DDG, there is a distinct vertex for each occurrence of a statement in the execution trace. The main issue of this approach is that the size of the graph can become too large. In another work, Kamkar [67] presents an algorithm which is based on graph reachability in dependence graphs. During the program execution, its execution trace is recorded and based on that, an execution tree with dependence information is constructed. Then, the slice is computed by traversing the execution tree backwards with respect to the dependencies. Zhang et al. [146] proposed three new, more precise algorithms for dynamic slicing, the most efficient of which is called the *Limited Processing* (*LP*) algorithm where the dynamic dependence graph is constructed on-demand in response to dynamic slicing requests from the execution trace that

is saved on the disk. While this approach greatly reduces the size of the dynamic dependence graph held in memory, the on-demand construction of the dynamic dependence graph is quite slow since it requires repeated traversals of the trace stored on the disk. Wang and Roychoudhury [129] present JSlice which performs backward dynamic slicing of sequential Java programs. Since backward slicing requires storing the execution trace, JSlice performs online compression during trace collection. The compressed trace representation is then traversed without decompression during slicing. Finally, in another work, Wang and Roychoudhury [130] present the concept of hierarchical dynamic slicing. The main idea of this work is to present the slice to the programmer hierarchically and in different phases instead of the whole slice all at once. This helps the programmer to understand large slices better and focus on what she needs.

Compared to static slicing, dynamic slicing has the benefit of producing smaller slices. This allows easier localization of bugs in the process of debugging applications. Although dynamic slicing was originally proposed for program debugging, it has been used for other purposes such as program comprehension [74], software testing [68], and software maintenance [71, 75] as well. Interested readers can refer to [76] and [118] for a more complete survey of different slicing techniques and their applications.

It is worth mentioning that the API trace slicing technique which will be introduced in the next chapter is completely different from the traditional dynamic program slicing approaches discussed here. As will be described later, API trace slicing works with traces of interactions between applications and frameworks, while traditional dynamic slicing techniques work with traces of program statements. Moreover, while in program slicing the dependency among the events is defined in terms of data and control dependencies, in API trace slicing, it is defined in terms of the use of common objects as targets, parameters, or return values of calls.

3.7 Summary

This chapter provided an overview of the most related work in the areas of framework documentation, framework usage comprehension, specification mining, concept location, and aspect mining. Although this literature review showed that a significant progress has been made in all these areas of research, none of them can completely address the issue of concept location for the purpose of framework comprehension.

Framework documentation approaches can explicitly document how to implement a target concept on top of a framework. However, their main difficulty is that they may become outdated as the framework evolves and manual effort is required for creating them. Consequently, many of the existing software frameworks suffer from the lack of good documentation. To address this issue, framework usage comprehension techniques are proposed that benefit from example applications of

the framework. Nonetheless, they are mainly beneficial for the understanding of fine-grained framework extensions, such as how to call a specific framework method or how to instantiate a particular framework class, but less so for coarse-grained concepts which may span multiple framework classes such as those which are of interest in this dissertation. Specification mining methodologies mainly mine for behavioral and temporal protocols that a program must follow when interacting with the API of a library. Therefore, they are not that useful for concept location in the context of framework comprehension. Existing concept location tools also do not focus on framework API usage and, thus, the code identified using these tools will still include many application-specific instructions that are irrelevant from the viewpoint of framework usage. Finally, aspect mining approaches aim to identify unknown crosscutting concerns to modularize them into aspects, and hence are not beneficial for the purpose of this dissertation.

This dissertation aims to address the above challenges in existing work and move forward the state of the art. To this goal, next chapter describes the details of the FUDA framework comprehension technique.

Chapter 4

The FUDA Framework Comprehension Technique

The review of existing literature presented in Chapter 3 revealed that current concept location and framework comprehension techniques fail to provide a solution for the problem of concept location in the context of framework comprehension. To address this issue, this chapter introduces the notion of *concept implementation templates* and *FUDA* (*Framework API Understanding through Dynamic Analysis*), an approach to automatically extract such templates from traces of sample applications [55]. A concept implementation template is a Java-like code example demonstrating the *implementation steps* that are necessary to instantiate a given concept. In particular, an implementation template specifies which framework packages to import, framework classes to subclass, framework interfaces to implement, and framework operations to call. Such a template can be used as a concise summary of the necessary implementation steps and as a starting point to further investigate the concrete concept implementations in the sample applications.

In the remainder of this chapter, the running example is first introduced in Section 4.1. After that, the notion of implementation templates is presented in detail in Section 4.2. An overview of the FUDA template extraction approach is presented in Section 4.3 followed by its detailed description in Section 4.4. Finally, Section 4.5 summarizes this chapter.

4.1 A Running Example

As the running example throughout this chapter, Figure 4.1 presents the code implementing a context menu using the JFace framework. The result of this code will look like the screenshot presented in Figure 2.3. The menu is located in `SampleView`, which is a visual component that displays trees using a `TreeViewer` (l. 36). The code was generated using one of Eclipse's wizards. The lines implementing the context menu are marked by •. The lines marked by ◦ implement a Welcome

```
     ...
  35 public class SampleView extends ViewPart {
  36   private TreeViewer viewer;
  37   private DrillDownAdapter drillDownAdapter;
 •38   private Action action1;
 •39   private Action action2;
 ○40   private WelcomeWindow welcomeWindow;
       ...
  98   class ViewContentProvider
  99     implements IStructuredContentProvider, ITreeContentProvider {
         ...
 162   }
 163   class ViewLabelProvider extends LabelProvider {
         ...
 189   }
 190   public void createPartControl(Composite parent) {
 ○191     welcomeWindow = new WelcomeWindow();
 ○192     welcomeWindow.open();
 193     viewer = new TreeViewer(...);
 194     drillDownAdapter = new DrillDownAdapter(viewer);
 195     viewer.setContentProvider(new ViewContentProvider());
 196     viewer.setLabelProvider(new ViewLabelProvider());
 197     viewer.setInput(getViewSite());
 •198     makeActions();
 •199     hookContextMenu();
 200   }
 •201   private void hookContextMenu() {
 •202     MenuManager menuMgr = new MenuManager("#PopupMenu");
 •203     menuMgr.setRemoveAllWhenShown(true);
 •204     menuMgr.addMenuListener(new IMenuListener() {
 •205       public void menuAboutToShow(IMenuManager manager) {
 •206         SampleView.this.fillContextMenu(manager);
 •207       }});
 •208     Menu menu = menuMgr.createContextMenu(viewer.getControl());
 •209     viewer.getControl().setMenu(menu);
 •210     getSite().registerContextMenu(menuMgr, viewer);
 •211   }
 •212   private void fillContextMenu(IMenuManager manager) {
 •213     manager.add(action1);
 •214     manager.add(action2);
 •215     manager.add(new Separator());
 •216     drillDownAdapter.addNavigationActions(manager);
 •217     manager.add(new Separator(IWorkbenchActionConstants.MB_ADDITIONS));
 •218   }
 •219   private void makeActions() {
 •220     action1 = new Action() {
 221       public void run() { showMessage("Action 1 executed"); }
 •222     };
 •223     action1.setText("Action 1");
 •224     action1.setToolTipText("Action 1 tooltip");
 •225     action2 = new Action() {
 226       public void run() { showMessage("Action 2 executed"); }
 •227     };
 •228     action2.setText("Action 2");
 •229     action2.setToolTipText("Action 2 tooltip");
 •230   }
       ...
 267 }
```

Figure 4.1: Implementation of a sample Eclipse view with a context menu (•)

```
 1 import org.eclipse.jface.action.Separator;
 2 import org.eclipse.jface.viewers.Viewer;
 3 import org.eclipse.jface.action.Action;
 4 import org.eclipse.jface.action.MenuManager;
 5 import org.eclipse.swt.widgets.Menu;
 6 import org.eclipse.jface.resource.ImageDescriptor;
 7 import org.eclipse.jface.action.IMenuListener;
 8 import org.eclipse.swt.widgets.Control;

 9 public class AppMenuListener implements IMenuListener {                    (l. 204)→
10   public void menuAboutToShow(menuManager) {                               (l. 205)→
11     Separator separator = new Separator(String)||(); //REPEAT          (l. 215, l. 217)→
12     menuManager.add(separator)||(appAction); //REPEAT            (l. 213-l. 215, l. 217)→
13   }
14 }

15 public class AppAction extends Action {                                    (l. 220, l. 225)→
16 }

17 public class SomeClass {
18   public void someMethod() {
19     Viewer viewer = ...;
20     Control control = viewer.getControl(); //MAY REPEAT                   (l. 208, l. 209)→
21     AppAction appAction = new AppAction(); //MAY REPEAT                  (l. 220, l. 225)→
22     appAction.setText(String); //MAY REPEAT                              (l. 223, l. 228)→
23     appAction.setToolTipText(String); //MAY REPEAT                       (l. 224, l. 229)→
24     MenuManager menuManager =
25       new MenuManager(String)||(String,String)||();                       (l. 202)→
26     menuManager.setRemoveAllWhenShown(boolean);                           (l. 203)→
27     AppMenuListener appMenuListener = new AppMenuListener();              (l. 204)→
28     menuManager.addMenuListener(appMenuListener);                         (l. 204)→
29     Menu menu = menuManager.createContextMenu(control);                   (l. 208)→
30   }
31 }
```

Figure 4.2: A sample implementation template for the concept "context menu"

window. They were manually added as an example of code that is completely unrelated to the context menu. The constituent parts of the view are created in `createPartControl()`. In particular, this method calls `makeActions()` and `hookContextMenu()`, which together create the context menu. In general, a context menu consists of one or more actions (l. 220, 225) and potentially one or more separators (l. 215, 217). It is constructed by a menu manager (l. 202, 208) and invoked by a menu listener (l. 204). The latter implements the callback method `menuAboutToShow()` (l. 205), which is called by JFace when the user clicks to open the context menu.

The context menu example illustrates some of the challenges in locating concepts in code. The implementation of the menu is scattered across and tangled with the implementation of the view; and it involves a complex interaction of several objects, namely view, menu manager, menu listener, menu, actions, and separators. To complicate matters, a concept implementation may also be scattered across several classes as in the case of Eclipse's drag&drop. Consequently, although locating a concept in the GUI of a sample application may be easy, locating its implementation in the application code is often challenging and time consuming.

4.2 Concept Implementation Templates

An implementation template generated automatically using the FUDA technique for the context menu example is illustrated in Figure 4.2. This template was generated using two traces that were collected by invoking the context menu in two sample applications: `SampleView` (Figure 4.1) and `Console`, which is part of Eclipse. The generated template in Figure 4.2 has the form of a tutorial-like example in Java-based pseudocode indicating the *implementation steps* necessary to instantiate the given concept. More specifically, an implementation template specifies the following implementation steps:

- Packages to import (l. 1–8 in Figure 4.2);
- Framework classes to subclass (l. 15);
- Interfaces to implement (l. 9);
- Methods to implement (l. 10);
- Objects to create (*e.g.*, l. 11, 21, 25, 27); and
- Methods to call (*e.g.*, l. 12, 20, 22–23, 26, 28–29).

Note that the specified steps involve only the elements of the framework API. For example, the method calls `makeActions()` and `hookContextMenu()` in Figure 4.1 are specific to that particular implementation and are not reflected in the template. The involved elements may be entirely framework-defined, *e.g.*, the implementation of `Separator`, which is instantiated in line 11, resides in framework code. Alternatively, the elements may also reside in the application code, provided that they are *framework-stipulated* (i.e., declared in the framework and implemented in the application code). For example, `AppAction` is both defined (l. 15) and instantiated (l. 21) in the application code; however, JFace's design stipulates the creation of subclasses of the framework-defined class `Action` in the application code. In addition to the basic implementation steps, the template also reflects:

- Call nesting, *e.g.*, `add()` is called directly or indirectly by `menuAboutToShow()` (l. 12);
- Call order, *e.g.*, menu listener is added to the menu manager (l. 28) before creating the menu (l. 29);
- Parameter passing patterns, *e.g.*, the control object passed to the menu creation method (l. 29) is obtained by a prior call to `getControl()` (l. 20);
- The comments `REPEAT` and `MAY REPEAT` indicate that the commented step appeared more than once in every or some of the traces used to generate the template, respectively.

Templates are rendered in ordinary Java with two exceptions:

- *Use the notion of '||' to present alternative argument types*: The templates use a special syntax to show that a method with a given name was called with different argument types. For example, `add(separator)||(appAction)` (l. 12 in Figure 4.2) is due to multiple calls to `add()` with different arguments (l. 213 and 215 in Figure 4.1). As another example, `new Separator(String)||()` (l. 11) illustrates different argument types with which the class `Separator` can be instantiated.
- *All variables are global:* What appears to be a local variable declaration in Java, such as `appAction` (l. 21), actually has global meaning in the template. For that reason, `appAction` can be used as a method argument in another method scope (l. 12).

A template extracted by FUDA is an approximation of the necessary implementation steps, and it can be incomplete or unsound or both. In particular, implementation steps can be missing (*false negatives*) or unrelated steps (*false positives*) can be present in some cases (see Chapter 5). Given two or more traces, FUDA will filter out any steps that are not common to all traces. If a necessary implementation step, say component registration, can be achieved in more than one way, such as by calling different methods, such step may get filtered out. Furthermore, FUDA relies on the assumption that input traces show the execution of the concept of interest in different contexts. For example, `SampleView` and `Console` provided entirely different contexts for the context menu concept. However, a view using a `TableViewer`, even though graphically different from our `SampleView`, which uses `TreeViewer`, would also call `setContentProvider` and `setLabelProvider`. As a result, a template generated from the latter two applications would have included these two JFace calls, even though they are not related to the context menu. Finally, some implementation details maybe absent in a template. For example, whereas the calls in lines 21–23 are marked as candidates to be repeated, the template does not reflect the fact that they should be repeated as a block, rather than individually. Nevertheless, the user can still extract the missing details from the actual sample code and the traceability links between the template and the application code, as shown in Figure 4.2, can support this task. In particular, each traceability link in this figure illustrates the mapping between the implementation step in the template and its corresponding program statements in the sample application presented in Figure 4.1.

4.3 The FUDA Approach Overview

Before delving into the details of FUDA, this section provides an overview of this technique. The key idea of this technique is to take advantage of the information

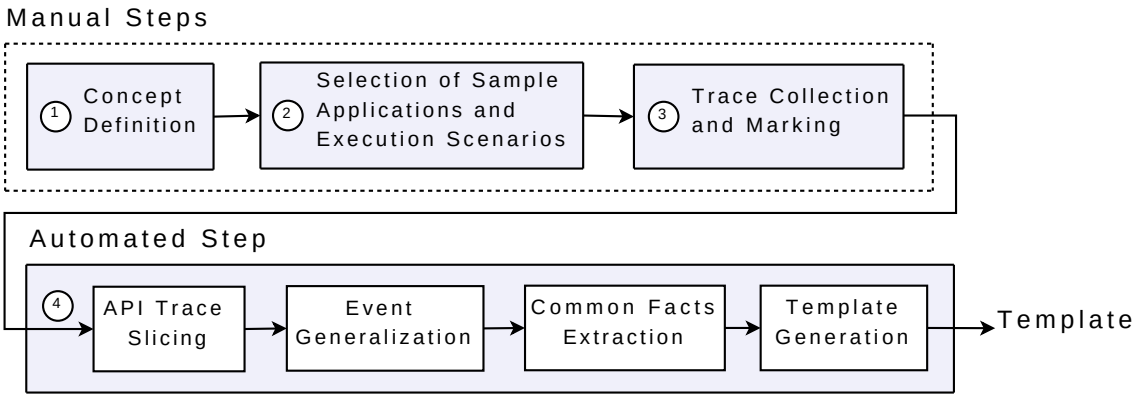


Figure 4.3: FUDA technique overview

contained in sample applications and utilize dynamic analysis to help determine the portions that are relevant to the implementation of the desired concept. To make the approach attractive in practise, the goal is to be able to extract useful implementation templates by using only a few sample applications. To do so, from the user's perspective, FUDA consists of four main steps (Figure 4.3) from which steps 1-3 are performed manually and step 4 is done automatically:

Step 1: *Concept Definition.* As the first step, the developer has to exactly specify what the concept of interest is, for which she wants to generate an implementation template.

Step 2: *Selection of Sample Application(s) and Execution Scenarios.* With respect to the concept of interest, the user needs to identify one or more sample applications that execute the desired concept in different contexts. A single application may already satisfy this goal, *e.g.*, an application implementing a context menu in two different views would have been sufficient in our example. As will be discussed later in the evaluations of template extraction (Chapter 5), the more the contexts differ across the executions, the lower the possibility of false positives.

Step 3: *Trace Collection and Marking.* In this step, each sample application is executed according to the execution scenarios designed in the previous step and a *tracer tool* collects a trace of all calls that occur at the boundary between the application and the framework API. For the purpose of pinpointing the location of the concept execution in the trace, the user informs the tracer tool of the moments right before and after the invocation of the concept, a process named *marking* [108].

Step 4: *Automatic Fact Extraction and Template Generation.* This step takes two or more collected traces and generates the template. This step mainly consists of (i) API trace slicing, (ii) event generalization, (iii) extracting facts about calls that are common across the traces, and (iv) generating the template from the common facts.

The following section elaborates on each step in more detail.

4.4 The FUDA Approach

Section 4.3 provided an overview of the FUDA framework comprehension technique and generally described what steps are followed from the user's perspective to generate concept implementation templates for a given concept. This section presents a detailed description of these steps.

4.4.1 Concept Definition

In FUDA, a *framework-provided concept* is defined as a unit of functionality which is accessible through the framework's API and is implemented in applications by writing *framework completion code*, *i.e.*, the framework provides a certain functionality and prescribes steps for its reification and configuration in applications. As mentioned earlier in Section 4.2, a typical framework completion code consists of framework API method invocations, class extensions, interface implementations, and so on. Therefore, a framework-provided concept may be implemented by one or more classes.

FUDA imposes two requirements around the concepts it can successfully analyze. First, the developer must have a minimal knowledge about the framework that provides the concept. FUDA requires as input the name of the framework's package so that the tracer tool can understand the framework's boundary and concentrate only on the interactions that cross this boundary. Without this delimitation, the analysis would be flooded with irrelevant information. In our running example (Section 4.1), the developer must know that those context menus are provided by JFace, and that JFace's package is named `org.eclipse.jface`.

Second, it must be possible to invoke the concept of interest in sample applications from their graphical or programmatic interfaces. The concept can itself be a graphical interface concept, such as the context menu, but it can also be any other concept, as long as it could be explicitly invoked from the sample applications' graphical or programmatic user interface. FUDA can produce implementation templates covering the entire life cycle of a concept, which involves concept creation (creating and setting up its implementation objects), concept invocations (calling the objects) and concept destruction (tearing down and disposing the objects). The following *concept-defining question* asks for the entire life cycle of a concept: "*How does one implement a context menu in an Eclipse view?*" Alternatively, FUDA can also produce implementation templates covering individual concept invocations, as exemplified by this question: "*What events occur when a user clicks on a figure?*"

4.4.2 Selection of Sample Applications and Execution Scenarios

It is necessary that users have access to one or more sample applications executing the concept of interest in different contexts. The assumption underlying the

FUDA approach is that modern object-oriented frameworks typically come with a number of example applications from which application developers can learn about frameworks' APIs. Moreover, thanks to many open source applications available on the Internet (*e.g.*, `http://www.sourceforge.net`) and code searchers (*e.g.*, `http://www.google.com/codesearch`), it is usually possible to find some sample applications implementing a desired concept. As it will be shown later in Chapter 5, in our experiments with twelve realistic concepts on top of four widely used frameworks, it was possible to use FUDA to retrieve quality implementation templates with only two sample applications.

Following the selection of sample applications, the user requires to design concept-invoking execution scenarios. The applications and the scenarios should be selected to achieve one or more of the following goals: (i) The scenarios are concept-focused: ideally the majority of the executed instructions are part of the concept. (ii) The concept is invoked separately from others as part of the scenario and the invocation can be explicitly marked. (iii) Each concept instance is invoked in a different context. A single application may already support the third goal, e.g., an application implementing a context menu in two different views would suffice. Because FUDA works by intersecting traces of different executions, the more the contexts differ, the lower the possibility of false positives. For the same reason, it is important to select scenarios that contain a similar variant of the concept, which minimizes false negatives. For example, if a variant of the context menu concept with a separator is desired, scenarios that contain separators should be selected. To design these execution scenarios, the user can benefit from anything that could be a source of information such as existing users of the applications, help systems of the applications, their user interfaces, manuals and documentation, existing test cases, and so on.

4.4.3 Trace Collection and Marking

As mentioned earlier, object-oriented software frameworks provide their services through their APIs. Therefore, to understand how an application uses a framework, one must focus on the interactions between the application and the framework API, *i.e.*, the calls from the application to the framework API and callbacks from the framework API to the application. Consequently, FUDA is interested in the interactions between the sample applications and the framework API.

Figure 4.4 illustrates the whole process of trace collection and marking. In this phase of FUDA, the user executes each sample application assisted by a tracer tool and invokes the concept of interest according to the execution scenarios designed in the previous step. If possible, pinpointing the moments before and after the concept invocation will improve the template extraction results, which is in fact essential for concepts whose defining question deals with the response to an event. For the context menu example, it amounts to instructing the tracer to *mark* subsequent events right before opening the menu and instructing it to stop marking

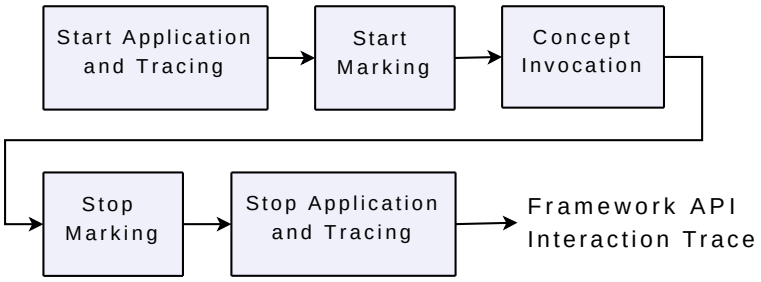


Figure 4.4: Process of trace collection and marking

right after the menu is open. If the moments before and after concept invocation cannot be pinpointed, the whole trace is marked. Concepts invoked through a programmatic interface can use the tracer tool to indicate the begin and end of the concept execution. The user has to also specify for the tracer tool the package(s) in which the framework of interest and the sample application reside, *e.g.*, `org.eclipse.jface.*` and `SampleView` for the context menu example. The tracer logs all the calls that occur at the boundary between the application and the framework, which results in a *framework API interaction trace*, called *API trace* for short. An atomic interaction collected by the tracer is called a *framework API interaction event* or *interaction event* for short, which is a runtime event corresponding to method or constructor calls executed at the boundary between the framework and application code. Each event has one of two directions:

- *Outgoing*: An event is *outgoing* if the call is made from within the application code and the target is either implemented in the framework code or it is framework-stipulated and implemented in the application code. An instance method that is implemented in an application class is framework-stipulated if it is declared in the framework. A constructor that is implemented in an application class is framework-stipulated if the class is a subtype of a framework class or interface.
- *Incoming*: An event is *incoming* (*i.e.*, a callback) if the call is made from within the framework code and the target is framework-stipulated and is implemented in the application code.

We consider both static and instance variants of methods. This definition reflects the direction of the interaction with respect to the application, and distinguishes events that cross the application-framework boundary from internal events. For instance, an outgoing event is triggered by the application, but is prescribed by the framework (or at least declared). This definition includes both: (i) targets fully implemented by the framework, such as a constructor or method, and (ii) targets prescribed by the framework and implemented in the application, such as an interface method implementation or an overridden method, provided that the framework declared it. An incoming event occurs when the framework is the driver

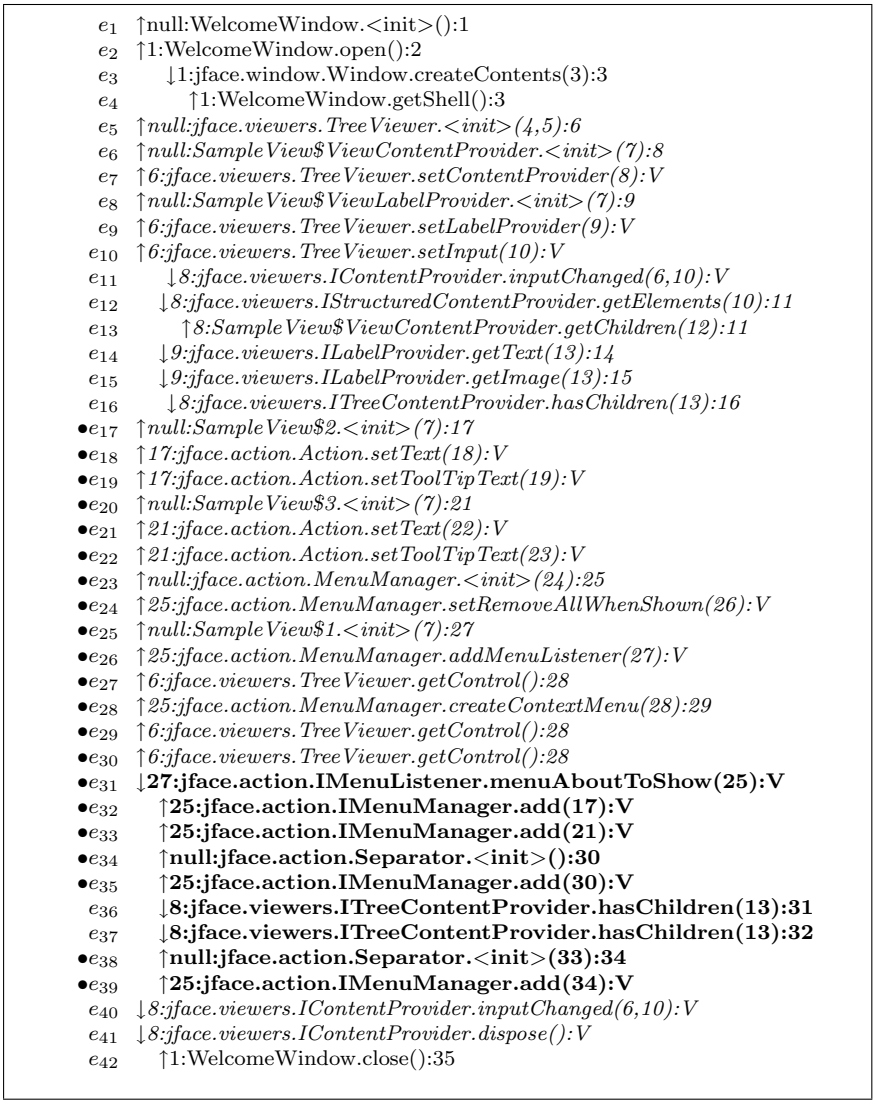


Figure 4.5: A framework API interaction trace resulted from executing the SampleView presented in Figure 4.1 and invoking the context menu

and the implementation is in the application. This direction is usually implemented by method callbacks, but also includes reflective usage of application objects by the framework.

The complete API trace produced by running `SampleView` from Figure 4.1 and invoking its context menu is shown in Figure 4.5. Each event is represented by the notation *D O:n(P):R*, where *D* represents the direction of the event, with "↓" denoting incoming and "↑" denoting outgoing events; *O* is the target object's ID[1] or "null" for constructor and static method calls; *n* represents the fully qualified name [42] of the target method or constructor; *P* is a list of IDs of objects passed as parameters; and *R* is the ID of the returned object or "V" if the return type

[1]At runtime, each object is assigned a unique hash value which we call the object ID.

is `void`. For brevity, the package prefix `org.eclipse` was removed for all JFace events. The events in bold face are those that were marked by informing the tracer about the moments before and after the context menu was invoked. This portion of the API trace is referred to as the *marked region*.

Most of the events in Figure 4.5 can easily be traced back to their corresponding code lines in Figure 4.1. The events e_1–e_{30} are generated when the method `createPartControl()` is called, *e.g.*, e_1 is due to line 191. The actual call to `createPartControl()` is not traced because it resides in `org.eclipse.ui`, which is not part of JFace. The calls in l. 209 and l. 210 are not traced for the same reason. Indentation denotes *event nesting*. For example, events e_3 and e_4 were generated in the control flow of the call to `welcomeWindow.open()` (l. 192). Anonymous classes are denoted by numbers separated from their host classes by `$`, *e.g.*, e_{17} constructs `action1` (l. 220).

The marked events (in bold face) were generated by the callback to `menuAboutToShow()`, which is called by JFace when a menu is being opened. That method calls `fillContextMenu()`, which generates the nested events e_{32}–e_{39}. The incoming events e_{36}–e_{37} are generated by method called in l. 216, which is not part of JFace and thus not traced. The last three events, e_{40}–e_{42}, are generated during cleanup (code not shown).

4.4.4 Automated Trace Processing

The automated trace processing stage receives two or more of the collected traces and generates a concept implementation template. It consists of the following steps.

API Trace Slicing

We have seen that the interaction events that occurred during the concept execution are marked by the user during the trace collection stage. The marked region therefore contains the events that were explicitly involved in running the concept, potentially among other irrelevant events. However, if the goal is to understand the complete life cycle of the concept, it is necessary also to consider calls related to the initialization and clean-up of the involved objects, which are not necessarily reflected in this marked region. For instance, in Figure 4.5, the marked region (bold faced) contains the events that occurred when the context menu was being opened. However, the events e_{17}–e_{22} create and initialize the context menu's actions, but they are not included in the marked region in the interaction trace. Likewise, the marked region may miss cleanup events, such as service deregistration.

To identify such relevant events, a novel slicing technique named *API trace slicing* is applied here. The key idea in this technique is to introduce the notion of *object dependency* between events. It is then used to compute the transitive closure of the events in the marked region to determine the *API trace slice*. In API trace slicing, the notion of dependency is based on the following heuristic:

Heuristic. Two events e_i and e_j in the API trace are said to be dependent if they involve at least one object in common.

The rationale behind this heuristic is motivated by common API usage patterns. For example, two method invocations sharing the same target object could be related by the fact that one invocation initializes the target for the second invocation, or the second invocation cleans up the object that was used in the first invocation. Similarly, an invocation that returns an object that is later used as a target or parameter in a subsequent invocation may be an invocation to a factory method. Based on this heuristic, the notion of object dependency can be defined in the following way:

Definition. Object-Dependent. Two events e_i=D_i O_i:$n_i(P_i)$:R_i and e_j=D_j O_j:$n_j(P_j)$:R_j are called to be *object-dependent* iff $((\{O_i, R_i\} \cup P_i) \cap (\{O_j, R_j\} \cup P_j)) \setminus \{\text{null}, \text{V}\} \neq \emptyset$, *i.e.*, they share any target, parameter, or returned objects.

With respect to this definition, one can introduce fine-grained versions of dependencies, based on how the common object is used in the events. For instance, since events are ordered in the trace, we can say that events e_i and e_j have a *Target-Parameter* (TP) dependency if e_i precedes e_j in the trace and the same object used as a target in event e_i is subsequently used as a parameter in event e_j. In this way, it is possible to define nine kinds of fine-grained dependencies between two events e_i and e_j such that e_i precedes e_j in the trace:

- *Target-Target* (TT) dependency if $O_i = O_j$;
- *Target-Parameter* (TP) dependency if $O_i \in P_j$;
- *Target-Return* (TR) dependency if $O_i = R_j$;
- *Parameter-Target* (PT) dependency if $O_j \in P_i$;
- *Parameter-Parameter* (PP) dependency if $P_i \cap P_j \neq \phi$;
- *Parameter-Return* (PR) dependency if $R_j \in P_i$;
- *Return-Target* (RT) dependency if $R_i = O_j$;
- *Return-Parameter* (RP) dependency if $R_i \in P_j$; and
- *Return-Return* (RR) dependency if $R_i = R_j$.

Note that in above definitions the object IDs must not be null or void. Careful consideration of these definitions indicates that all these dependency types except RR represent some form of object passing from the first event to the second event. The purpose of RR will be explained later in Section 4.4.4. For example, in Figure 4.5, some events outside the marked region that contribute to the implementation of the context menu and have at least one kind of object dependency to at

least one of the events in the marked region are: e_{28} has TT dependency to e_{32}, e_{18} has TP dependency to e_{32}, e_{26} has PT dependency to e_{31}, e_{17} has PP dependency to e_{25}, e_{23} has RT dependency to e_{39}, and e_{23} has RP dependency to e_{31}.

Based on the above notion of object dependency, the notion of *object-relatedness* can be introduced as the transitive closure of object dependency. Then, an API trace slice is defined in the following way:

Definition. API Trace Slice. An API trace slice is a portion of the input trace consisting of all the marked events and the unmarked events that are object-related to the marked events.

Based on this definition, those events that do not depend in any way on an event of the marked region are declared irrelevant and are removed from the trace. Note that this definition of API trace slicing is based on the assumption that all the events in the marked region are relevant for the concept of interest which is not necessarily true. However, as will be shown later in Chapter 5 for realistic concepts, these false positives are typically filtered out in the common fact extraction phase (see Section 4.4.4).

In Figure 4.5, the unmarked events that are object-related to the marked ones are typeset in italic font. For instance, e_5 is object-dependent to e_7 through the object with ID 6, and e_7 is object-dependent to e_{36} through object 8. Consequently, e_5 is object-related to the marked event e_{36} and thus part of the slice. Note that slicing eliminates the steps implementing the Welcome window (e_2–e_4, e_{42}), which are unrelated to the context menu. The resulting sliced trace that does not include the unrelated events is illustrated in Figure 4.6.

API trace slicing is an approximation of the actual dependencies between API calls. However, the approximation worked perfectly for the real framework APIs in our template extraction evaluation (see Chapter 5): there was not a single false negative due to slicing. Moreover, as mentioned earlier, the false positives are also typically filtered out later based on the commonalities across traces. Slicing is optional since some concepts focus on the invocation only, in which case no slicing is needed and only the marked events are further processed. For example, the user may explicitly ask the question of what code is involved when a context menu is opened. In this case, slicing should be disabled through an option before the traces are processed and FUDA will only process the marked regions. If marking is not used at all during the trace collection, FUDA will process the entire trace, which may be used, for example, to identify framework calls made on start-up and shutdown of an application. Additionally, the API trace slicing technique can be configured to account for all types of dependencies or for just a subset, depending on the user's opinion and nature of the concept (see prototype implementation of FUDA in Section 5.2).

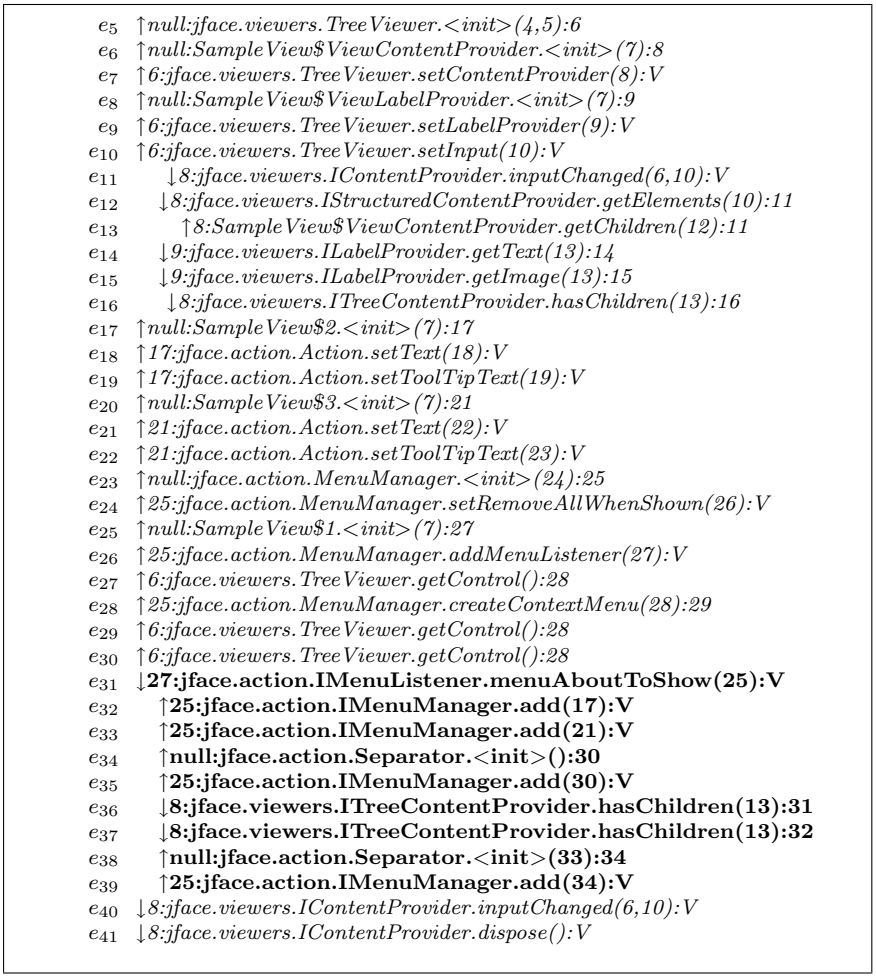


Figure 4.6: The sliced trace resulted from the API trace presented in Figure 4.5

Event Generalization

At this point of the analysis, the sliced traces still contain application-specific information. Since the next step of the analysis will attempt to find commonalities among traces, it is necessary to abstract any application-specific elements from the events. This will allow the next processing stage to compare traces in terms of framework API types. To this aim, event generalization replaces the application-specific names of events with appropriate framework names[2]. For example, the fully qualified name of e_6 in Figure 4.6, *i.e.*, `SampleView$ViewContentProvider.<init>`, is application-specific and event generalization replaces it by `[jface.viewers.IStructuredContentProvider, jface.viewers.ITreeContentProvider].<init>`. The two names in brackets refer to the framework interfaces that are implemented by `ViewContentProvider` (l. 99).

To do this generalization, a simple static analysis on the type hierarchy of the

[2]Acknowledgments: This part of the work is mostly done by my colleague Thiago Tonelli Bartolomei.

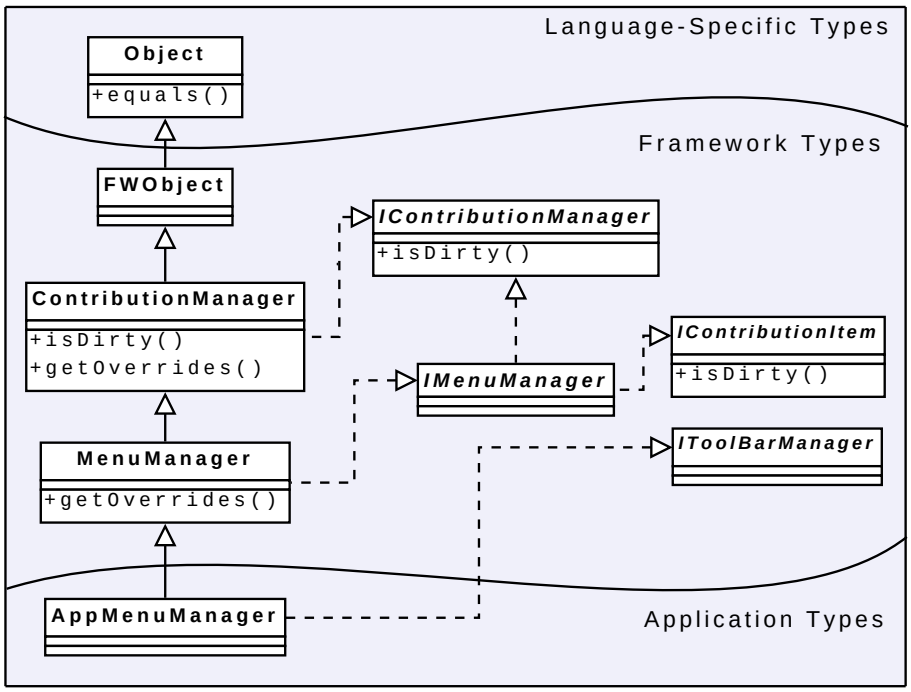


Figure 4.7: Boundaries of application, framework, and language-specific types

target's enclosing class is performed to find the so called *root types* of the event and generate a generalized event for each root type. Thus, a single event may potentially result in several generalized events. Event generalization treats calls to instance methods, constructors, and static methods differently. In the following, the generalization rules are discussed for each type of target. This explanation is done using Figure 4.7 that shows the context menu implementation in JFace, but further extended to assist in explaining the process.

Instance Methods. When generalizing an instance method call, the procedure aims at maximum generality and searches for the topmost types that declare the method. For example, the method `equals` in Java is declared by `Object` and although the method may be overridden in many subclasses, it conceptually belongs to `Object`. A method may also have multiple topmost types, *e.g.*, generalization of a call to `AppMenuManager.isDirty()` would identify two topmost types: `IContributionManager` and `IContributionItem`, because both interfaces declare the method.

Constructors. A constructor invocation means that an object of a certain class is going to be created. However, an application class may specialize many framework and application-specific types. Looking bottom-up from the target type, the type hierarchy is a directed acyclic graph (DAG), as can be seen in Figure 4.7 for the class `MenuManager`. For constructor invocations, a meaningful generalization is to generate one root type for each type at the bottom borderline of the framework of

the target type's inheritance DAG. The rationale is that, both for incoming and outgoing events, the construction of objects with application types is irrelevant, hence we look for the most specific framework type. The analysis traverses the DAG bottom-up and generates a root type for the first framework-declared type for each branch. For example, if `AppMenuManager` is the target, it generates `MenuManager` and `IToolBarManager` as root types; if `MenuManager` is the target, the only root type will be `MenuManager` itself. Note that another approach could be to generate a root type for each topmost framework type. While this approach would help agglomerate more events, we would lose the information of framework variants in use. To exemplify the problem, consider the hierarchy in the figure, where the framework defines a root class called `FWObject`. Using this approach would yield `FWObject` for every constructor call to a subclass of a framework class.

Static Methods. Although a static method cannot be polymorphically called, it can be hidden by an equally-named static method in a subclass. Thus, the procedure searches the type hierarchy of the application class being instantiated and returns the first type that declares the method. For example, in Figure 4.7 both `MenuManager` and `ContributionManager` declare the `getOverrides()` static method. Depending on which class is used statically, a different method is really being used. Therefore, in order to find the actual member in use we search the type hierarchy of the target bottom-up and assume the first type that declares the member as the root type.

As an example of generalized traces, Figure 4.8 illustrates the generalized trace for the sliced trace presented in Figure 4.6.

Common Facts Extraction

The aim of this phase is to identify the facts that are common among all input generalized traces. As it will be described in the following of this section, three types of facts are extracted from each generalized trace: *event occurrence facts*, *event nesting facts*, and *event dependency facts*. The first represents the occurrence of interaction events in the generalized trace, while the other two represent the existence of certain relationships among events. Then, *common facts* are computed as intersections of the extracted fact sets across all generalized traces. Figure 4.9 presents different kinds of common facts for the concept context menu extracted from two generalized traces for `SampleView` (Figure 4.8) and `Console` example applications.

Event Occurrence Facts. Event occurrence facts, called *event facts* for short, are the names of the methods and constructors that were called at the application-framework boundary and the corresponding call directions (Figure 4.9(a)). They abstract away the numbers of occurrences, object IDs, and parameter and return types of the corresponding calls. The rationale is that two methods with the same

e_5 ↑null:[jface.viewers.TreeViewer].<init>(4,5):6
e_6 ↑null:[jface.viewers.IStructuredContentProvider, jface.viewers.ITreeContentProvider].<init>(7):8
e_7 ↑6:[jface.viewers.ContentViewer].setContentProvider(8):V
e_8 ↑null:[jface.viewers.LabelProvider].<init>(7):9
e_9 ↑6:[jface.viewers.ContentViewer].setLabelProvider(9):V
e_{10} ↑6:[jface.viewers.Viewer].setInput(10):V
e_{11} ↓8:[jface.viewers.IContentProvider].inputChanged(6,10):V
e_{12} ↓8:[jface.viewers.IStructuredContentProvider].getElements(10):11
e_{13} ↑8:[jface.viewers.ITreeContentProvider].getChildren(12):11
e_{14} ↓9:[jface.viewers.ILabelProvider].getText(13):14
e_{15} ↓9:[jface.viewers.ILabelProvider].getImage(13):15
e_{16} ↓8:[jface.viewers.ITreeContentProvider].hasChildren(13):16
•e_{17} ↑null:[jface.action.Action].<init>(7):17
•e_{18} ↑17:[jface.action.IAction].setText(18):V
•e_{19} ↑17:[jface.action.IAction].setToolTipText(19):V
•e_{20} ↑null:[jface.action.Action].<init>(7):21
•e_{21} ↑21:[jface.action.IAction].setText(22):V
•e_{22} ↑21:[jface.action.IAction].setToolTipText(23):V
•e_{23} ↑null:[jface.action.MenuManager].<init>(24):25
•e_{24} ↑25:[jface.action.IMenuManager].setRemoveAllWhenShown(26):V
•e_{25} ↑null:[jface.action.IMenuListener].<init>(7):27
•e_{26} ↑25:[jface.action.IMenuManager].addMenuListener(27):V
•e_{27} ↑6:[jface.viewers.Viewer].getControl():28
•e_{28} ↑25:[jface.action.MenuManager].createContextMenu(28):29
•e_{29} ↑6:[jface.viewers.Viewer].getControl():28
•e_{30} ↑6:[jface.viewers.Viewer].getControl():28
•e_{31} ↓27:[jface.action.IMenuListener].menuAboutToShow(25):V
•e_{32} ↑25:[jface.action.IContributionManager].add(17):V
•e_{33} ↑25:[jface.action.IContributionManager].add(21):V
•e_{34} ↑null:[jface.action.Separator].<init>():30
•e_{35} ↑25:[jface.action.IContributionManager].add(30):V
e_{36} ↓8:[jface.viewers.ITreeContentProvider].hasChildren(13):31
e_{37} ↓8:[jface.viewers.ITreeContentProvider].hasChildren(13):32
•e_{38} ↑null:[jface.action.Separator].<init>(33):34
•e_{39} ↑25:[jface.action.IContributionManager].add(34):V
e_{40} ↓8:[jface.viewers.IContentProvider].inputChanged(6,10):V
e_{41} ↓8:[jface.viewers.IContentProvider].dispose():V

Figure 4.8: The generalized trace resulted from the sliced trace illustrated in Figure 4.6

name but different parameter or return types or number of parameters are likely to be conceptually equivalent within an API, *i.e.*, they probably serve the same purpose and differ only in their input data. An event fact $D_i\ t_i.n_i$, where t_i is a type name, is extracted from a generalized trace iff the trace contains one or more events $D_i\ O{:}[\ldots,t_i,\ldots].n_i{:}R$, where O is any object ID or "null" and R is any object ID or "V". We say that the events *match* such an event fact. For example, a_2 is extracted from the generalized trace due to its events corresponding to e_{18} or e_{21} in Figure 4.8.

The events in Figure 4.8 that match the common event facts in Figure 4.9(a) are marked by •. The remaining events are effectively filtered out as they were unique to this trace.

Event Nesting Facts. Event nesting facts record the calling context for outgoing calls (Figure 4.9(b)). A nesting fact $a_i{\rightarrow}a_j$, where a_i and a_j are event facts, is

a_1	↑jface.action.Action.<init>
a_2	↑jface.action.IAction.setText
a_3	↑jface.action.IAction.setToolTipText
a_4	↑jface.action.MenuManager.<init>
a_5	↑jface.action.IMenuManager.setRemoveAllWhenShown
a_6	↑jface.action.IMenuListener.<init>
a_7	↑jface.action.IMenuManager.addMenuListener
a_8	↑jface.viewers.Viewer.getControl
a_9	↑jface.action.MenuManager.createContextMenu
a_{10}	↓jface.action.IMenuListener.menuAboutToShow
a_{11}	↑jface.action.Separator.<init>
a_{12}	↑jface.action.IContributionManager.add

(a) Common event occurrence facts

$a_{10} \rightarrow a_{11}$
$a_{10} \rightarrow a_{12}$
jface.action.IMenuListener$\rightarrow a_{11}$
jface.action.IMenuListener$\rightarrow a_{12}$

(b) Common nesting facts

RT(a_1, a_2)	RT(a_1, a_3)	RP(a_1, a_{12})	TT(a_2, a_3)
TP(a_2, a_{12})	TP(a_3, a_{12})	RT(a_4, a_5)	RT(a_4, a_7)
RP(a_4, a_{10})	RT(a_4, a_{12})	RT(a_4, a_9)	TT(a_5, a_7)
TT(a_5, a_9)	TP(a_5, a_{10})	TT(a_5, a_{12})	RP(a_6, a_7)
RT(a_6, a_{10})	TT(a_7, a_9)	PT(a_7, a_{10})	TP(a_7, a_{10})
TT(a_7, a_{12})	RP(a_8, a_9)	TP(a_9, a_{10})	TT(a_9, a_{12})
PT(a_{10}, a_{12})	RP(a_{11}, a_{12})		

(c) Common dependency facts

Figure 4.9: Common facts

produced whenever the generalized trace contains two events e_k and e_l such that (i) e_k and e_l match a_i and a_j, respectively; (ii) e_l is outgoing; and (iii) e_l is directly nested in e_k in the trace. For every such fact, an additional nesting fact of the form $t_i \rightarrow a_j$, where t_i is the type name of a_i, is also produced. The rationale for generating the nesting facts in the form of $t_i \rightarrow a_j$ can be described using Figure 4.10. This figure presents two code snippets for two different Java applets on top of the `Applet` framework. In the first one the framework's method `showStatus()` is called from within the framework's overridden method `start()` and in the second one, the `showStatus()` method is called from within the framework's overridden method `stop()`. If we just generate nesting facts in the form of $a_i \rightarrow a_j$, then the fact that both of the sample Java applets called the `showStatus()` method would have been missed in the set of common nesting facts because of different calling contexts. Hence, to not lose the information that both sample applets called the `showStatus()` method, in addition to generating nesting facts in the form of $a_i \rightarrow a_j$, nesting facts in the form of $t_i \rightarrow a_j$ are generated as well.

Event Dependency Facts. Event dependency facts represent call sequence and object passing patterns. As mentioned in Section 4.4.4, nine types of event dependencies are considered in concept trace slicing. Analogously, nine types of dependency facts are extracted from each generalized trace (Figure 4.9(c)): target-

```
import java.applet.Applet;
public class SampleApplet1 extends Applet {
   ...
   public void start() {
       ...
       showStatus(``Applet Started!'');
       ...
   }
   ...
}
```

(a)

```
import java.applet.Applet;
public class SampleApplet1 extends Applet {
   ...
   public void stop() {
       ...
       showStatus(``Applet Stopped!'');
       ...
   }
   ...
}
```

(b)

Figure 4.10: Two sample Java applets calling the `showStatus()` method in two different contexts

target (TT), target-parameter (TP), target-return (TR), parameter-target (PT), parameter-parameter (PP), parameter-return (PR), return-target (RT), return-parameter (RP), and return-return (RR). Similar to the definitions provided in Section 4.4.4, a target-target dependency fact $\mathrm{TT}(a_i, a_j)$, where a_i and a_j are event facts, is produced whenever the generalized trace contains two events e_k and e_l such that (i) e_k and e_l match a_i and a_j, respectively; (ii) e_k precedes e_l in the trace; and (iii) both e_k and e_l have the same object as target. The analogous definitions for the remaining dependency fact types are obtained by modifying the third condition. For example, if the return object ID of e_k is used as a parameter in e_l, the resulting dependency fact type is $\mathrm{RP}(a_i, a_j)$. Dependency facts indicate sharing of objects and object passing; e.g., PR and TR may represent the registration of an object with a framework and subsequent retrieval.

After the common facts are computed, the event facts that originated from the same generic events (because of multiple type names due to generalization) are collapsed and the affected common nesting and dependency facts are updated accordingly.

Template Generation

This section provides the details of template generation algorithms. The pseudocode of these algorithms is provided in Appendix B. The input to this phase are the three sets of common facts extracted from the generalized traces in the previous step and the generalized traces themselves. The common facts determine the overall structure of the template, and the generalized traces are used to extract

additional details as needed. The template generation phase mainly executes the following steps:

1. The constituent classes of the template are identified.
2. For each of the identified classes, its constituent constructor and methods are determined.
3. The program statements for the constructor and each of the methods are identified.
4. The supertypes of each class are specified.
5. The class names and variable names are determined.
6. The variable names are broadcasted based on common dependency facts.
7. The package names are removed from the fully qualified names of types and the list of package imports are determined.

The above steps are discussed in more detail in the following sections. However, the reader should bear in mind that some technical details are omitted for simplicity.

Create Classes. The first step in generating a template is to identify its constituent classes. To this aim, the common incoming method calls and outgoing constructor calls are used. An incoming method call means that it should be implemented in an application class. An outgoing constructor call which is of interest in this phase happens when an application class extends or implements one of the framework classes or interfaces.

For this purpose, first the set of incoming method calls are retrieved from the set of common event facts. Then, a class `C` is created for each group of incoming method calls that are related by TT dependencies in the set of common dependency facts. If two incoming method calls have the same target then it means that they belong to the same application-side class. For instance, the class in l. 9 (Figure 4.2) is created for the fact a_{10} (Figure 4.9) which does not participate in any TT dependencies and thus forms its own group.

After creating a separate class for each group of incoming method calls that have TT dependencies, the list of outgoing constructor calls are retrieved from the set of common event facts. Since no object yet exists when a constructor call happens, the target object ID of that constructor call is null. However, the object ID of the newly created object is returned in the returned object of that constructor call event. Therefore, if an outgoing constructor call has RT dependency to any of the incoming method calls grouped in a class `C` or RR dependency to any of the outgoing constructor calls grouped in that class, that constructor call is also grouped in that class. For instance, the constructor call a_6 is assigned to the class

in l. 9 (Figure 4.2) due to the dependency fact RT(a_6,a_{10}). If there does not exist such a class and the constructor call is calling an interface or an abstract class (*e.g.*, when instantiating anonymous classes), then it means that one of the application classes should implement or extend that interface or abstract class. Therefore, a new class is created and that constructor call is assigned to it. For example, the class in l. 15 is created for a_1, a call to the constructor of the abstract class `Action`. If none of these situations happen, it means that the outgoing constructor call is a call to one of the framework classes and therefore no classes need to be created.

Create Methods and Constructors. For each incoming method call assigned to a class in the previous step, a method is created in that class. For example, the method in l. 10 (Figure 4.2) is created because of the incoming method call a_{10} (Figure 4.9) assigned to the class in l. 9 (Figure 4.2). A constructor is created in a class if nesting facts whose source is any of the constructor calls assigned to that class are present. In our running example, we do not have an example of this.

Create Statements. The sets of common event facts and common nesting facts are consulted to identify the statements that should go into the body of each method. Three cases can happen:

1. For every method `m` of each class `C`, based on common nesting facts, the set of outgoing event facts that are in the calling context of that method are set as its statements. For example, the nesting fact $a_{10} \rightarrow a_{12}$ (Figure 4.9(b)) places the call in l. 12 (Figure 4.2).

2. Based on the common nesting facts, the set of outgoing event facts that are in the calling context of class `C`, but are not in the calling context of any of its methods, trigger the creation of a specific method named `someMethod` in class `C` to host those outgoing event facts. For instance, if the nesting fact $a_{10} \rightarrow a_{12}$ did not exist, the third nesting fact in Figure 4.9(b) would have triggered the creation of the method `someMethod()` in `AppMenuListener` and the call to `add()` would have been placed there. As another instance, consider the example about Java applets presented in Figure 4.10. If we wanted to generate a template for Java applets using the sample applets presented in that figure, the call to `showStatus()` method would have been placed in the `someMethod()` method.

3. A particular class named `SomeClass` with a `someMethod` method is created to host the set of common outgoing event facts for which no calling contexts are specified by the nesting facts. For example, the set of event nesting facts in Figure 4.9(b) does not identify any calling contexts for the outgoing event facts a_1-a_9 and hence, all of them are placed in the method `SomeClass.someMethod()` in Figure 4.2.

After specifying the event facts (statements) that should go into the body of each method, the generalized traces are consulted to identify how many times each event fact is repeated in a calling context. The statements are then commented in the following ways:

- `REPEAT`: If the event fact is called multiple times in the given calling context in all the input generalized traces. For example, l. 12 (Figure 4.2) is commented as `REPEAT` since a_{12} was called multiple times in every trace in the calling context of a_{10}.
- `MAY REPEAT`: If the event fact is called multiple times in some of the input generalized traces, but not all of them (*e.g.*, l. 20).

Finally, within each method, calls are sorted in an order determined by the dependency facts (except RR facts, which are only used in class creation). More specifically, the call order of the statements are obtained by applying the topological sort on a graph with statements as nodes and the dependency facts (except RR facts) as directed edges among them. For example, the call order in the context menu template was determined in this way.

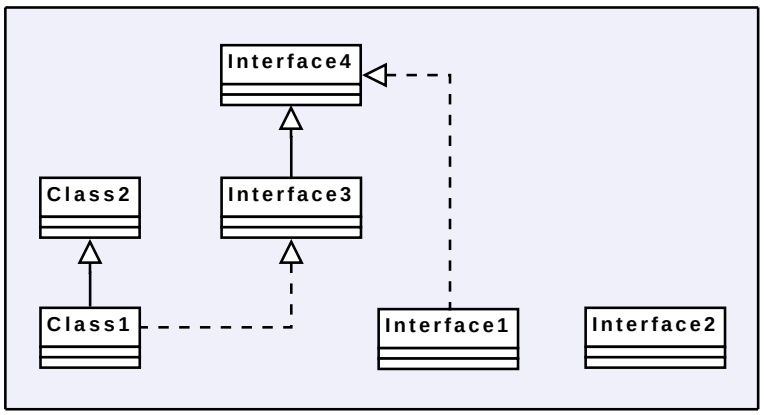
Identify Supertypes. Superclass and interfaces for each class (except `SomeClass`) are determined by constructing a type hierarchy of target types of method and constructor calls assigned to it in the first step of the template generation algorithm. Then, the leaves of this type hierarchy identify the interfaces and the superclass of the given class. Since in Java each class can have only one superclass, therefore, at most one leaf of the type hierarchy would be of type class or abstract class and the rest of the leaves would be interfaces. For instance, Figure 4.11(a) illustrates the set of incoming method and constructor calls assigned to an imaginary class `C` and Figure 4.11(b) illustrates an imaginary type hierarchy built based on these calls. In this imaginary example, the superclass for the class `C` would be `Class1` and the interfaces would be `interface1` and `interface2` since they are at the bottom-most level of the type hierarchy. As another example consider Figure 4.7 in which there are no interfaces for the class `AppMenuManager`, but its superclass is `MenuManager`.

Generate Class and Variable Names. The aim of this step is to generate class and variable names. In other words, this phase converts the statements that so far are in the form of event facts into more Java-like formats. For this purpose, the following steps are performed:

1. *Identifying the names of the application classes (except `SomeClass`).* Each class is named by pre-pending `App` to its superclass name or to one of its interface names if no superclass is present.

```
↑ Class1.<init>
↑ Interface1.<init>
↑ Interface2.<init>
↓ Class2.m2
↓ Interface3.m3
↓ Interface4.m4
```

(a) Incoming method calls and outgoing constructor calls assigned to class `C`.



(b) Type hierarchy of calls assigned to class `C`

Figure 4.11: An imaginary type hierarchy for framework types assigned to an imaginary class `C`

2. *Identifying the return type and parameters of each method (except `someMethod()`) in every class.* For this purpose, we benefit from the method signature declared in the framework API. If the method has only one declared signature, then the return type and parameters of this method are set as what is declared in the framework API. Otherwise, if the method has several signatures, then different alternatives are shown using the '||' notation as described in Section 4.2.

3. *Identifying the return type, return type variable name, and parameters for each statement in each method.* Each statement in the body of each method can be either a constructor or method call. Similar to the previous step, also in this phase, the declared signatures of constructor and method calls in the framework API are used. Again, if there are several signature declarations in the framework API for a given constructor or method call, the notation of '||' is used to show different alternatives.

   More specifically, if the statement is a constructor call, the return type is the same as the target type of the constructor call and the return type variable name equals the target type in which the first letter is in lower case. If the statement is a method call, the return type and parameters are set as what is declared in that method's signature. The return type variable name equals the return type, but the first letter is in lower case.

Broadcasting Variables. The goal of this phase is to show how objects are passed among the program statements in the template. Using a graph in which program statements are set as nodes and the dependency facts (except RR facts) as directed edges among them, the variables defined in the statements in the previous step are broadcasted to their successor nodes. For instance, in Figure 4.2, `appAction` is passed as a parameter to `add` in l. 12 because of RP(a_1,a_{12}). Finally, parameter objects of framework-stipulated types that were not returned by any other calls are provided by dummy declarations as in l. 19.

Identifying Imports. This is the last step of the template generation algorithm in which the package names are removed from the fully qualified names of types in the generated template and the list of package imports is created. In particular, the package names are removed from the fully qualified names of the following things and are added to the list of package imports: (1) the supertypes of each class, (2) the return type of each method, (3) the parameter types of each method, (4) the return types of statements, (5) the target of statements, and (6) the parameters of statements. In addition, the package names are removed from the fully qualified names of the following things, but they are not added to the package imports list: (1) If the type is one of the language-specific types such as `java.lang.String` or `java.lang.Object`, it is not required to add `java.lang` to the list of package imports. and (2) The method names in each class. Each method in the class either implements or overrides one of the methods declared in that class's supertypes. Hence, it is not required to add the package names removed from the method names to the list of imports because the package names of those supertypes are already added to the imports set.

4.4.5 Existing Issues in Template Generation

This section discusses some of the remaining issues in the template generation algorithms presented in the previous section and need to be addressed in future work. These issues are related to the problems with creating classes, their methods and constructors, and the cyclic dependencies among the program statements in the body of a method. Nevertheless, the developer can still extract the information missing in the templates because of these issues from the sample applications using the traceability links in the templates.

Issues in Generating Classes and Their Methods

As discussed in Section 4.4.4, the current approach for generating classes and their methods is based on the dependencies among the incoming method calls and outgoing constructor calls. Nonetheless, this approach can be problematic in some situations:

```
import FrwkCls;
public class AppCls extends FrwkCls {
    public void m() { // framework-stipulated method
        // Does Something!
    }

    public static void main(String[] args) {
        AppCls appCls = new AppCls();
        appCls.m();
    }
}
```

(a) A simple code snippet

```
↑null:[FrwkCls].<init>():1
↑1:[FrwkCls].m():V
```

(b) Generalized trace

```
↑FrwkCls.<init>
↑FrwkCls.m
RT(↑FrwkCls.<init>, ↑FrwkCls.m)
```

(c) Common facts

```
import FrwkCls;
public class SomeClass {
    public void someMethod() {
        FrwkCls frwkCls = new FrwkCls();
        frwkCls.m();
    }
}
```

(d) Template generated by FUDA

Figure 4.12: An example showing the issues of outgoing calls to framework-stipulated methods and constructors

- No methods are created in the template for framework-stipulated methods for which there are only outgoing calls, but not incoming calls in the trace. For instance, consider the simple code snippet presented in Figure 4.12(a) in which the application calls the framework-stipulated method `m`, but there are no incoming calls to this method as shown in the generalized trace (Figure 4.12(b)). Based on the common facts extracted from this trace (Figure 4.12(c)), FUDA generates the template illustrated in Figure 4.12(d). As can be seen in this figure, the fact that the application should override the method `m` is missed. One potential solution to this problem is to identify the calls to framework-stipulated methods that are implemented on the application side.

- No classes are created in the template for those that extend non-abstract framework-provided classes and for which there are no incoming calls to their methods. For example, consider again the example presented in Figure 4.12. In this example, class `AppCls` extends the framework's concrete class `FrwkCls`. However, since there are no incoming calls to framework-stipulated method `m`, FUDA did not reflect this fact in its resulting template, *i.e.*, the template does not include any classes extending the framework class `FrwkCls`. Again, one

possible solution to this problem is to identify calls to framework-stipulated methods.

- FUDA generates various classes for different instances of a single class from which different methods are called by the framework. For instance, suppose an application-side class `C1` implements framework methods `m1` and `m2`, both provided by a single framework type. Moreover, suppose there are two instances of `C1` such that one calls `m1` and the other calls `m2`. In this particular example, FUDA would generate two different classes, one implementing the method `m1` and the other one implementing the method `m2`, which is not necessarily appropriate. To address this issue, one strategy for creating the classes would be to group incoming method calls and outgoing constructor calls based on their types instead of their dependencies. Nevertheless, this approach can be problematic too. For example, consider the following example in which two different classes implementing the same interface `Interface1`, but the second one implements another interface `Interface2` as well:

  public class C1 **implements** Interface1 { ... }
  public class C2 **implements** Interface1, Interface2 { ... }

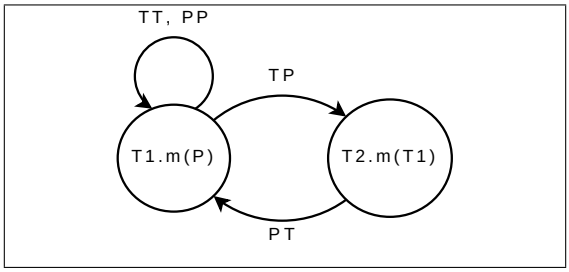
  In this example, if the method and constructor calls are grouped based on their types to form classes, then we will have two different classes, one implementing just the `interface1` and the other one implementing just the interface `interface2` which is not correct. Consequently, we decided to generate classes based on their dependencies. This approach at least ensures that incoming calls on a single instance will be treated together and the anomalies listed above can be treated by the developer and/or more sophisticated strategies are also possible. For instance, in the current prototype implementation of FUDA (Section 5.2), users have this option to merge classes if their supertypes overlap to form more complex classes.

Issue of Cyclic Dependencies among Method Statements

As mentioned in Section 4.4.4, within each method, the order of the statements are determined by applying a topological sort on a graph with statements (*i.e.*, event facts) as nodes and the dependency facts (except RR facts) as directed edges. However, it is possible that these dependency facts form some cycles in the graph built in this way. In this case, since it is impossible to apply topological sort on a cyclic graph, then it becomes impossible to order the method statements in this way. In this case, benefitting from more sophisticated techniques such as mining iterative patterns [84] might be of help. In the current prototype implementation of FUDA, the user is just warned of this fact using the comment `/* UNKNOWN ORDER FOR THE STATEMENTS */` and the method statements are listed in a random order.

```
...
T1.m(P);
T2.m(T1);
T1.m(P);
...
```
(a) Original code snippet

```
...
↑ ID_T1:[T1].m(ID_P):V
↑ ID_T2:[T2].m(ID_T1):V
↑ ID_T1:[T1].m(ID_P):V
...
```
(b) Generalized trace



(c) Dependency graph

```
...
/* UNKNOWN ORDER FOR THE STATEMENTS */
T1.m(P); // REPEAT
T2.m(T1);
...
```
(d) FUDA template

Figure 4.13: Issue of cyclic dependency among the statements within the body of a method

Figures 4.13-4.15 illustrate three situations in which dependency facts form cycles among the method statements. All these figures indicate the original code snippet in the sample application, the generalized trace, the dependency graph built out of the generalized trace, and the template generated based on that generalized trace. Note that in these figures, the nodes of the dependency graphs constitute the event facts and therefore they should not indicate the method parameters. However, they are illustrated for better understanding of the examples. The situations that are presented in these figures were actually observed in the evaluations of template extraction for real framework-provided concepts presented in Chapter 5.

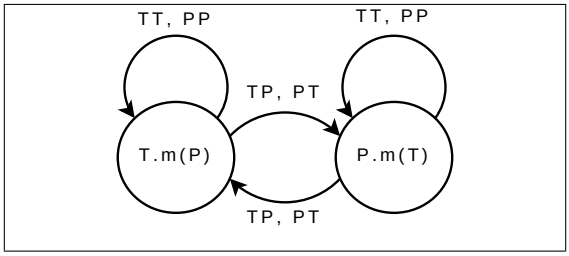
Figure 4.13 indicates a situation in which the statements in the body of a method form a dependency cycle. In this example, the second statement (*i.e.*, `T2.m(T1)`) has TP dependency to the first statement and PT dependency to the third statement while the first and the third statements are exactly the same (*i.e.*, `T1.m(P)`). Since in FUDA, multiple calls to a given method are collapsed into a single event fact, therefore there is a cycle in the dependency graph as shown in Figure 4.13(c).

```
...
for(int i = 1; i <= 2; i++) {
    T.m(P);
    P.m(T);
}
...
```

(a) Original code snippet

```
...
↑ ID_T:[T].m(ID_P):V
↑ ID_P:[P].m(ID_T):V
↑ ID_T:[T].m(ID_P):V
↑ ID_P:[P].m(ID_T):V
...
```

(b) Generalized trace



(c) Dependency graph

```
...
/* UNKNOWN ORDER FOR THE STATEMENTS */
T.m(P); // REPEAT
P.m(T); // REPEAT
...
```

(d) FUDA template

Figure 4.14: Issue of cyclic dependency among the statements because of a loop

Figure 4.14 indicates a situation in which the calls are in the body of a loop. The resulting trace of this loop is presented in Figure 4.14(b). As can be seen in this figure, although there is no cyclic dependency between the two statements in the body of the loop, since those statements are repeated in the trace, they form a cycle in the dependency graph.

Finally, Figure 4.15 presents a situation in which two different application-side classes are implementing the same interface and an instance of each of them is created for a desired concept. When FUDA generates a template using a trace of such a code snippet (*i.e.*, Figure 4.15(a)), the statements in the body of the same method (or constructor) in both of these classes are placed together and may form a cycle as presented in Figure 4.15(c). The reason for this situation is that all these statements are in the calling context of the same method (or constructor) based the set of common nesting facts and hence they are all put in the body of that method (or constructor). However, it should be noted that in FUDA, the aim was to show what statements were in the calling context of a given method, rather than

indicating alternatives of statements that may go into the body of that method. Representing these variabilities requires further research.

4.5 Summary

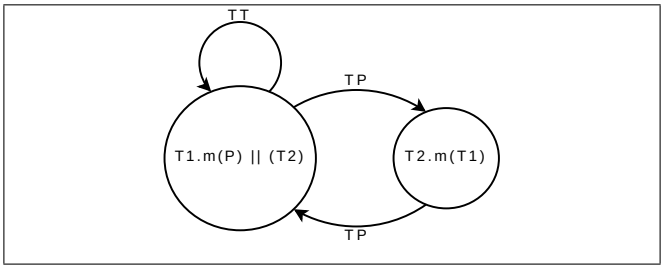
This chapter presented the notion of concept implementation templates as a Java-based pseudocode that indicates the necessary implementation steps to realize a desired concept on top of a given framework, such as which framework packages to import, interfaces to implement, classes to subclass, methods to implement, objects to create, methods to call, as well as some additional information such as call nesting, order of calls, and object passing patterns. The FUDA framework comprehension technique was also introduced as an automated approach for extracting such templates from traces obtained by invoking the concepts of interest in sample applications. The next chapter presents the prototype implementation of FUDA and will illustrate that it is possible to use FUDA to generate quality implementation templates with high precision and recall for real framework-provided concepts.

```
public class AppCls1
implements Interface1 {
    public void AppCls1() {
        T1.m(P);
        T2.m(T1);
    }
}
public class AppCls2
implements Interface1 {
    public void AppCls2() {
        T1.m(T2);
    }
}
...
AppCls1 appCls1 = new AppCls1();
AppCls2 appCls2 = new AppCls2();
...
```

(a) Original code snippet

```
↑ null:[Interface1].<init>():V
  ↑ ID_T1:[T1].m(ID_P):V
  ↑ ID_T2:[T2].m(ID_T1):V
↑ null:[Interface1].<init>():V
  ↑ ID_T1:[T1].m(ID_T2):V
```

(b) Generalized trace



(c) Dependency graph

```
public class AppInterface1
implements Interface1 {
    public void AppInterface1() {
        /* UNKNOWN ORDER FOR THE STATEMENTS */
        T1.m(P) || (T2); // REPEAT
        T2.m(T1);
    }
}
```

(d) FUDA template

Figure 4.15: Issue of cyclic dependency among the statements because of implementing the same method or constructor

Chapter 5

Template Extraction Evaluation

Chapter 4 introduced the notion of concept implementation templates and FUDA as an automated approach for the extraction of such templates from traces collected when the desired concept was invoked in a number of sample applications. This chapter presents the results of an empirical evaluation performed to assess the quality of templates generated using this technique for realistic framework-provided concepts. For this purpose, FUDA was prototyped as a tool for Java and this tool was used to generate implementation templates for a collection of twelve concepts for four popular frameworks, some of which were sampled from developers' forums. This study showed that the approach can produce templates with only very few false positives and false negatives for realistic concepts and sample applications.

The rest of this chapter proceeds as follows. Section 5.1 outlines the research questions and the hypotheses of this empirical study. Section 5.2 describes the prototype implementation of FUDA. Section 5.3 explains the setup of this empirical evaluation. Section 5.4 presents the evaluation results followed by Section 5.5 that discusses the threats to the validity of these results. Finally, Section 5.6 summarizes this chapter.

5.1 Experiment Objectives

This section presents the objectives for which this empirical study was designed and conducted.

5.1.1 Experiment Definition

The aim of the FUDA framework comprehension technique presented in Chapter 4 is to benefit from existing sample applications that implement a desired framework-provided concept to generate an implementation template for that concept. It was also discussed that an implementation template summarizes the steps necessary

to realize that concept such as which framework packages to import, framework classes to subclass, framework interfaces to implement, framework services to call, and so on. To assess how successful the FUDA approach is in extracting such templates, this experiment was designed and conducted to answer the following research questions:

1. Can FUDA generate quality implementation templates with high precision and recall?
2. Can FUDA generate quality implementation templates using only a few sample applications?
3. What is the impact of API trace slicing on the quality of implementation templates? How useful is it?

5.1.2 Hypothesis Formulation

Based on the research questions presented in the previous section, the following evaluation hypotheses are formulated for this study:

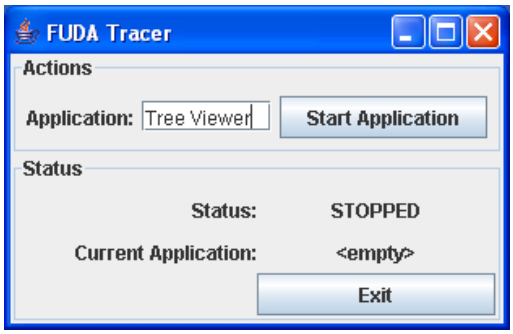
Hypothesis 1: FUDA can extract templates with high precision and recall from only two traces and/or two sample applications implementing the desired concept in two different contexts.

Hypothesis 2: API trace slicing improves the precision and recall of the templates generated using the FUDA technique.

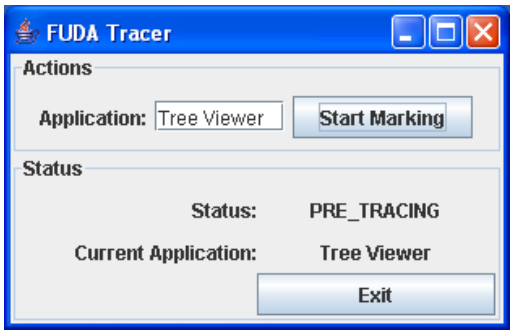
To make FUDA more practical, we aim to keep the number of traces per concept as small as possible since the selection of sample applications and collection of traces represent manual effort and the installation and execution of sample applications can be cumbersome too. Therefore, Hypothesis 1 was formulated based on using only two sample applications implementing the given concept in two different contexts. Nevertheless, it is worth mentioning that in an earlier experiment [53, 52] we showed additional applications cause templates to concentrate on the minimal common implementation steps, without much improvement in terms of false positives or negatives.

5.2 Prototype Implementation of FUDA

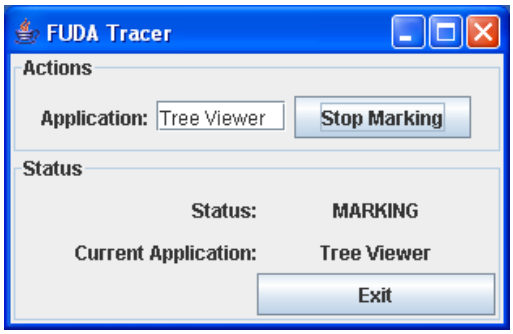
To test the hypotheses presented in Section 5.1.2, the FUDA technique was prototyped as an integrated tool for Java consisting of two parts [54]: *FUDA Tracer* and *FUDA Analyzer*. As described in Chapter 4, FUDA prescribes the use of a



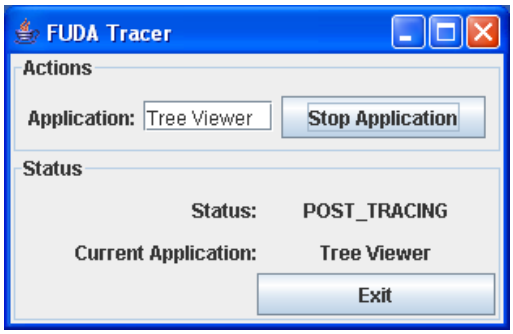
(a) User names the trace (*e.g.*, Tree Viewer). Pushes the *Start Application* button to put the FUDA Tracer in the recording mode. Then, opens the sample application.



(b) User clicks on the *Start Marking* button to put the FUDA Tracer in the marking mode. Then, invokes the desired concept in the sample application.



(c) When the job of the desired concept is done, user presses the *Stop Marking* button to finish marking the trace.



(d) User closes the sample application. Next, pushes the *Stop Application* button to stop recording. Then, exits from the FUDA Tracer by pressing the *Exit* button.

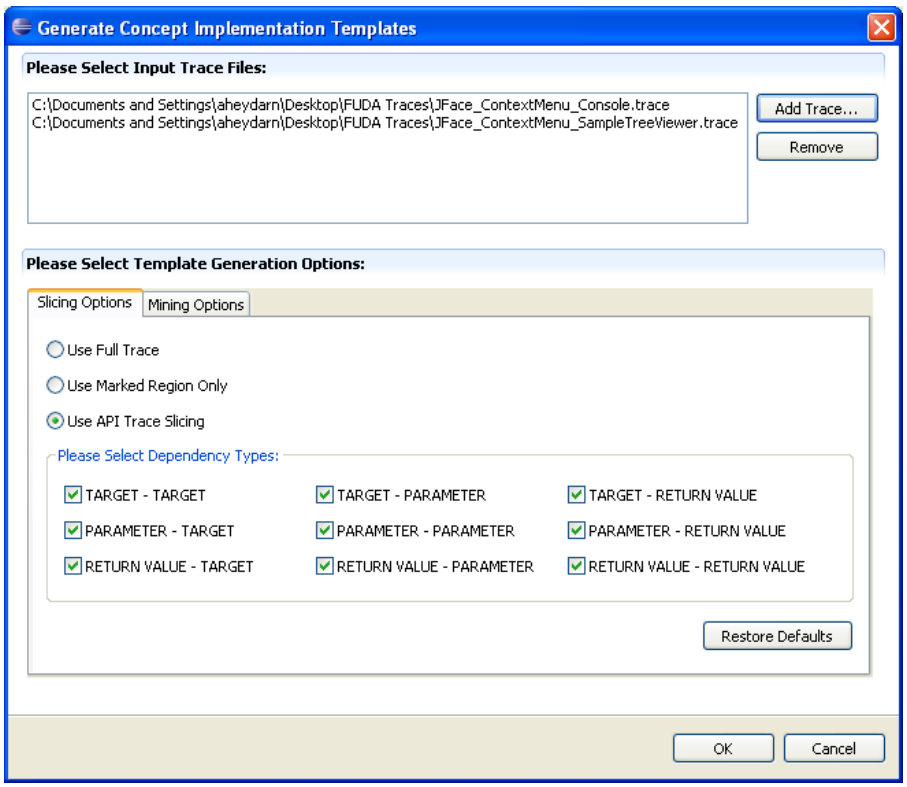
Figure 5.1: The FUDA Tracer GUI for recording an API trace

tracer and an analyzer tool. The former is responsible for assisting the developer in collecting and marking the traces during the trace collection step (discussed in Section 4.4.3), and the latter implements the automated analysis phase (described in Section 4.3). Currently, this prototype implementation contains 161 Java classes distributed in 49 packages and includes about 15,000 lines of code[1]. A flash demonstration of these tools can be obtained from [41].

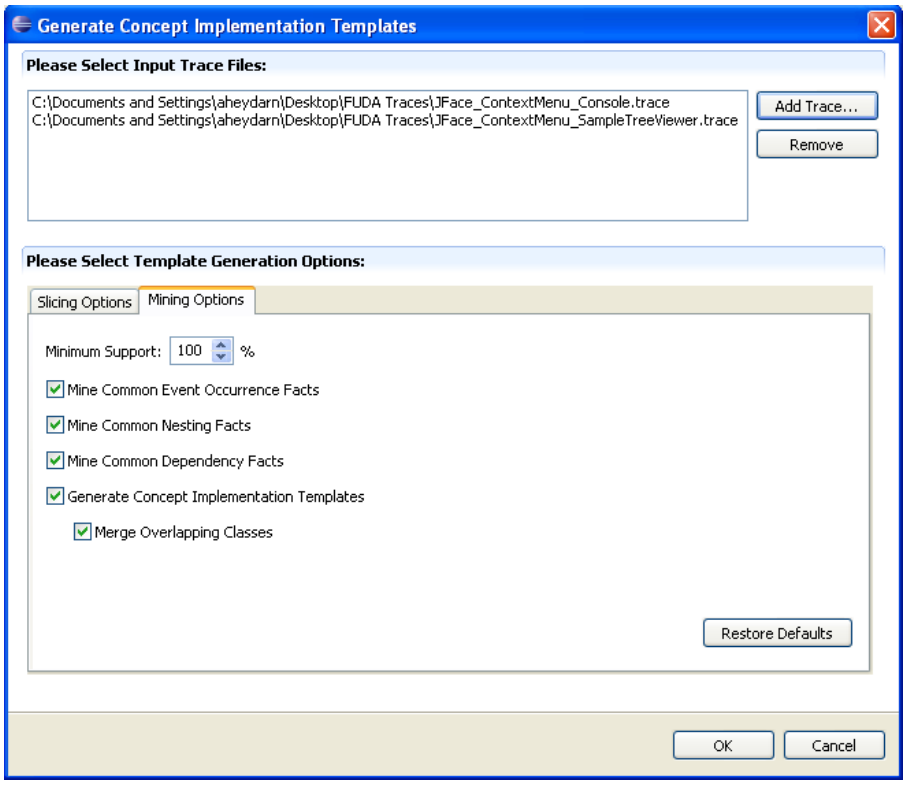
5.2.1 FUDA Tracer

The current prototype implementation of FUDA Tracer instruments applications using aspects written in *AspectJ*[2]. To instrument code inside the Eclipse platform, the FUDA Tracer benefits from the *AJEER*[3] plug-in. For this tracer, the user has to

[1]Excluding comments and blank lines.
[2]`http://www.aspectj.org/`.
[3]`http://ajeer.sourceforge.net/`.



(a) Slicing options



(b) Mining options

Figure 5.2: The FUDA Analyzer GUI for getting input traces and template generation options

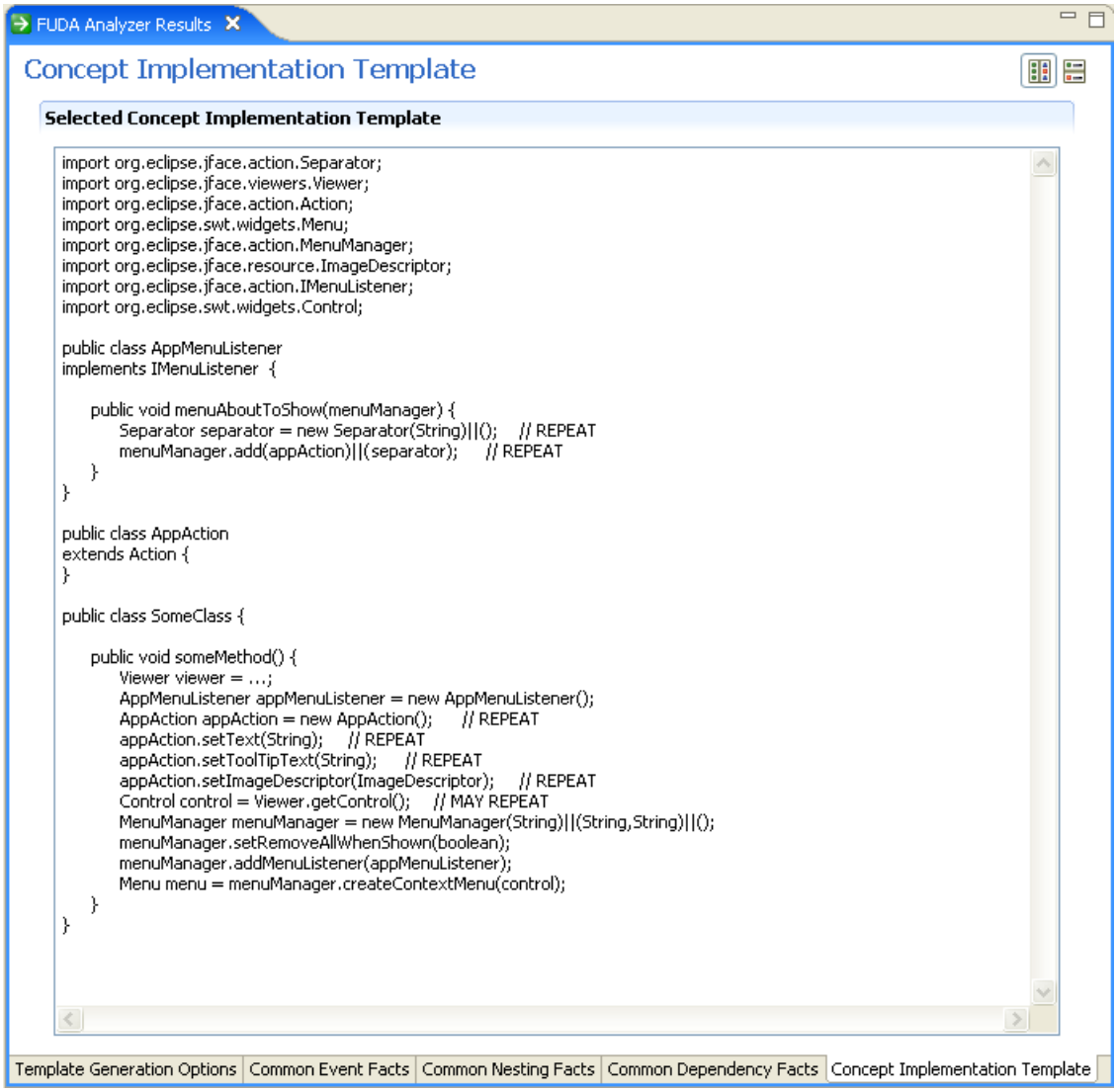


Figure 5.3: The FUDA Analyzer GUI for presenting the results of FUDA's automated analysis phase

specify the packages in which the framework of interest (*e.g.*, `org.eclipse.jface.*`) and the sample applications (*e.g.*, `ca.uwaterloo.gsd.SampleTreeViewer`) reside and then execute those instrumented sample applications under this tracer. The tool also contains a GUI that helps developers demarcate the begin and the end of a concept's execution as depicted in Figure 5.1. This figure also shows the process that the user must follow to record a trace using the FUDA Tracer.

5.2.2 FUDA Analyzer

FUDA Analyzer is implemented as a plug-in for Eclipse 3.2. The FUDA Analyzer accepts API traces stored in files by the FUDA Tracer tool, applies the whole

process of the automated analysis phase, and outputs the results. The tool also contains a GUI that helps developers to identify the input traces (Figure 5.2), different API trace slicing options (Figure 5.2(a)), and various template generation options (Figure 5.2(b)). Depending on the user's choices and the nature of the concept, the developer can use the slicing options to (i) disable the slicing to account for either the marked region only or the whole trace; or (ii) configure the slicing to account for all types of dependencies or for just a subset. Based on the template generation options selected, the tool can output only different kinds of common facts or it can generate complete concept implementation templates. The user also has the option to merge template classes whose supertypes overlap as discussed in Section 4.4.5. Finally, as described in Section 4.4.4, since this prototype implementation uses the *frequent closed itemset mining* [144] technique to find the commonalities among the input traces, the user has the option to determine the minimum support whose default value in this prototype is 100%, *i.e.*, only the implementation templates that are supported by all the input traces are generated. To conduct this mining, this prototype uses *LCM ver. 3.0* (*Linear time Closed itemset Miner*) [123] that outperforms many of the existing algorithms.

FUDA Analyzer also provides a GUI for presenting the analysis results (Figure 5.3). This GUI illustrates the selected options, different kinds of common facts extracted from the input traces, and the generated concept implementation template in separate tabs.

5.3 Experiment Setup

This section presents the design of this empirical study and the way it was set up to verify the hypotheses formulated in Section 5.1.2. In particular, this evaluation setup included the following steps:

1. Selection of frameworks on top of which the experiment concepts were chosen.
2. Selection of experiment concepts for which the templates were generated by applying the FUDA technique.
3. Selection of sample applications and execution scenarios to generate the templates.
4. Trace collection by using the selected sample applications and execution scenarios.
5. Specifying the quantitative and qualitative analysis procedures.

In the following, the details of these steps are explained in detail.

5.3.1 Selection of Frameworks

The evaluation includes four widely-used frameworks:

- *Eclipse*[4]— Eclipse is a large framework containing nearly two million lines of code [26] which is currently undergoing extensive development. The Eclipse project was originally created by IBM and is supported by a consortium of software companies and the open source community. The Eclipse framework is composed of many smaller frameworks that built upon each other. One of the main frameworks in Eclipse is the Eclipse Platform framework, a plug-in based framework that can be used to develop and integrate software tools such as Integrated Development Environments (IDEs). For instance, the popular Java Development Tool (JDT) framework which is widely used by programmers as a Java IDE is an extension of the Eclipse platform framework.

- *JFace*[5]— JFace is a user interface application framework which is built on top of the Standard Widget Toolkit (SWT). SWT provides the fundamental building blocks of a user interface in a typical Eclipse application. However, using JFace, the developer can implement user interface components with less code and effort than if she had started at the basic SWT widget level. In particular, it includes classes for handling common user interface components such as wizards, preference pages, actions, text, image and font registries, and dialogs. Although the JFace's heritage is based on the frameworks that are widely used for writing IDEs, most of JFace is generally useful in a broad range of graphical desktop applications. JFace is widely used, from simple stand-alone applications that do not use the Eclipse runtime, to workbench-based Rich Client Platform (RCP) applications, to Eclipse plug-ins.

- *Graphical Editing Framework*[6] (*GEF*)— The Graphical Editing Framework (GEF) is an open source framework that provides a visually rich, consistent graphical editing environment for developing applications on the Eclipse Platform. GEF is application neutral and provides the groundwork to build different kinds of applications, including, but not limited to, charts and graphs, reports, activity diagrams, GUI builders, UML diagram editors, and even WYSIWYG text editors for HTML. More specifically, GEF allows developers to create a rich graphical editor for existing application models. GEF employs an MVC (model-view-controller) architecture which enables simple changes to be applied back to the model originating from the view.

- *Java2D*[7]— The Java2D API is a framework for drawing two-dimensional graphics in Java, as well as an extension of the capabilities of Sun's Abstract Window Toolkit (AWT). The Java2D framework provides a robust package

[4] `http://www.eclipse.org/`.
[5] `http://wiki.eclipse.org/index.php/JFace`.
[6] `http://www.eclipse.org/gef/`.
[7] `http://java.sun.com/products/java-media/2D/`.

of drawing and imaging tools to develop elegant, professional, high-quality graphics. This framework provides extensive support for image compositing and alpha channel images, a set of classes to provide accurate color space definition and conversion, and a rich set of display-oriented imaging operators.

The main reasons for selecting these specific four frameworks are that they are widely used, they provide a variety of concepts, and sample applications for these frameworks are readily available in open source. Although they all involve graphical concepts, FUDA is also applicable to non-graphical frameworks too, as long as the concepts of interest can be explicitly invoked from the sample applications' graphical or programmatic user interface.

5.3.2 Selection of Concepts

Table 5.1 presents the concepts selected for this evaluation. The Eclipse and JFace concepts, except Focus, were selected as representative for FUDA based on the author's prior familiarity with these frameworks. The remaining concept defining queries were sampled from developer forums of the respective frameworks and FUDA steps were performed for them without much prior knowledge of the corresponding frameworks. In total, fourteen concepts were selected to cover a variety of characteristics. These characteristics include:

- *Scope.* A concept can be either in the scope of FUDA's intended usage or not. A concept is considered out of scope if it is very unlikely to find applications and scenarios satisfying the three goals from Section 4.4.2 for it.
- *Slicing.* It is not required to perform API trace slicing if the concept's definition includes only the events that happened in the marked region or spans the full trace.
- *Frequency.* A concept can be either frequent among the existing example applications of the framework or rare. A frequent concept can be also referred to as one of the framework's basic concepts which is typically implemented by most of its applications.
- *Complexity.* A concept can be either simple or complex in terms of implementation complexity measured as template size.
- *Atomicity.* A concept can be either composite in the sense of consisting of several variable subsets of implementation steps or atomic if only a fixed set of steps is involved.

In Table 5.1, an "X" in a column in front of a concept indicates that the concept has the corresponding characteristic in that column.

Table 5.1: The choice of frameworks, concepts, and sample applications for the template extraction evaluation (* indicates concepts from developer forums)

Framework	Concept	Defining Question	Properties					Sample Applications	
			In Scope	Slicing	Frequent	Simple	Composite	Name	Source
Eclipse	Table Viewer	How to implement a typical Eclipse table viewer?	X	-	X	-	X	Editor List	eclipse-plugins
								Table Viewer	Eclipse Wizard
	Tree Viewer	How to implement a typical Eclipse tree viewer?	X	-	X	-	X	LDAP Browser	eclipse-plugins
								Tree Viewer	Eclipse wizard
	Navigate	How to create the tree navigation buttons (i.e., *Go Home*, *Go Back* and *Go Into*) in a tree viewer's toolbar?	X	X	-	X	X	KTreeMap	SourceForge
								SVN Repository	Subclipse
	Focus*	What events happen when a user clicks on the title-bar of an Eclipse view?	X	-	X	X	-	LDAP Browser	eclipse-plugins
								Editor List	eclipse-plugins
JFace	Context Menu	How to implement a context menu in an Eclipse view?	X	X	X	X	X	Tree Viewer	Eclipse Wizard
								Console	Eclipse UI
	Toolbar Button	How to add a button to a view's toolbar?	X	X	X	X	X	Package Explorer	Eclipse JDT
								Crosscutting Comparison	AJDT
	Content Assist	How to implement a content assistant in an Eclipse text editor?	X	X	-	-	X	Java Editor	Eclipse JDT
								JSP Editor	Eclipse WTP
GEF	Select*	What events happen when a user clicks on a figure in a GEF editor?	X	-	X	X	-	Flow	GEF Examples
								Shapes	GEF Examples
	Figure*	How to implement drawing a figure in a GEF editor?	X	X	X	-	X	Flow	GEF Examples
								Shapes	GEF Examples
	Connection*	How to implement drawing a connection between two figures in a GEF editor?	X	X	X	-	X	Flow	GEF Examples
								Shapes	GEF Examples
	Title-bar Color*	How to change the color of a GEF editor's title-bar?	-	-	-	X	-	–	–
								–	–
Java 2D	Moving Shapes*	How to draw shapes and let the user drag them?	X	X	X	X	X	GTEditor	Google Code
								GeoSoft	Google Search
	Circle Drawing*	How to draw a red circle on a black background?	X	X	X	X	X	JHotDraw	SourceForge
								Scribble	Google Search
	Rounded Image*	How to make the corners of an image rounded?	-	-	X	X	-	–	–
								–	–

Slicing is optional, as an invocation scenario can sometimes span the full concept lifecycle, in which case the full trace is used. For example, Tree Viewer and Table Viewer are the concepts in Table 5.1, where slicing was not used since the scenario involving view opening and closing spanned the entire view lifecycle. Because of trace slicing, FUDA also works well for concepts having life-cycles spanning beyond the marked trace region, which are those shown with "X" in the Slicing column. We also included two concepts, Focus and Select, for which only the marked region is used, *i.e.*, the user was just interested in the events that happened when the given concept was invoked.

The FUDA technique could be applied for frequent concepts, in which case finding sample applications is likely easy. It may be applicable to rare concepts, too, if the user has already identified one or two applications with appropriate execution scenarios. For example, users may apply FUDA to find out the implementation of a rare concept that they noticed in an existing application. Also, concepts that may appear rare at first might not be rare after all. For example, the choice of red and black in Circle Drawing may be rare, but setting background and figure colors is not.

Most of the considered concepts are composite as they include variable parts. For example, a context menu may or may not include a separator. When applying FUDA, the user has to be aware that optional features will be lost if they are not present in both traces. Thus, if the user is interested in context menus with separators, both traces have to involve such menus. Concepts with only few implementation steps tend to be atomic and more complex ones are usually composite.

5.3.3 Selection of Sample Applications and Execution Scenarios

As mentioned earlier in Section 5.1.2, in this evaluation, we tried to keep the number of sample applications and/or execution scenarios per concept as small as two to make the FUDA technique attractive in practice. With respect to this, for each concept, two sample applications implementing the given concept in two different contexts were selected and for each of them one execution scenario was designed. Table 5.1 presents these sample applications as well as their sources. As can be seen in this table, the sample applications mainly came from the following different sources:

- Applications packaged with the framework itself such as example applications for the GEF concepts;
- Applications listed in online repositories such as `eclipse-plugins.org` or `SourceForge.net`;
- Applications that are part of a larger familiar environment such as Java Development Tools (JDT) or Eclipse WTP; or

- Applications that are automatically generated with the help of wizards such as Eclipse Wizard.

Selection of these applications involved the following strategies:

- Reliance on prior familiarity with a given application. This strategy was followed mainly for the Eclipse and JFace concepts because of author's prior knowledge of these frameworks;
- Browsing and running the standard examples of the framework. This strategy was mostly applied for the GEF concepts;
- Searching or browsing in online application repositories. For instance, GTEditor for the concept Moving Shapes was identified on Google Code by the search keyword "shape" and seeing a screenshot of a drawing editor; or
- Tips by others. For example, Eclipse WTP for the concept Content Assist was suggested by a colleague.

Selecting the applications for each concept took anywhere from no time for Eclipse JDT or wizards thanks to author's prior familiarity to up to an hour of searching and browsing for `eclipse-plugins.org` or `SourceForge.net`. The selection process had a significant learning effect: familiarity with framework-packaged examples or applications inspected for a given concept significantly reduced the selection time for the next concept. Some execution scenarios were already specified by the defining questions, *e.g.*, *"How does one draw a figure in a GEF editor?"* In other cases, an action invoking the concept of interest had to be identified, *e.g.*, opening action for Context Menu.

5.3.4 Trace Collection

The FUDA Tracer was used to collect the API traces. As discussed in Section 5.2.1, to trace each sample application, the user had to specify the packages in which the framework of interest and the sample application reside. As the packages of interest, `org.eclipse.*`, `org.eclipse.jface.*`, `org.eclipse.gef.*`, and `java.awt.*` were specified for Eclipse, JFace, GEF, and Java2D frameworks respectively. Note that only the calls at the application-framework boundary are traced, which are significantly fewer than all the calls involved in the implementation of a concept. As a result, API tracing is quite efficient. For example, tracing all of GEF was almost unnoticeable when using GEF applications. However, the applications ran two to three times slower when all of Eclipse was traced for Eclipse concepts. Collecting a single trace took anywhere from several seconds to a few minutes on a laptop with a single-core Pentium M 1.6MHz processor, 1GB of RAM, and Windows XP.

5.3.5 Template Generation

The templates were generated using the FUDA Analyzer introduced earlier in Section 5.2.2. To maximize the reach of the API trace slicing for all the concepts that require slicing, all types of dependencies in conjunction were used. The running time for this step on a laptop with a single-core Pentium M 1.6MHz processor, 1GB of RAM, and Windows XP, was anywhere from a few seconds up to six minutes. Based on the measurements done, the main bottleneck of the process was the number of the dependency facts being generated. However, performance optimization was not a goal in this prototype implementation since the running times for the evaluated concepts were satisfactory.

5.3.6 Analysis Procedure

This section describes the procedure followed for analyzing the generated templates in order to verify the hypotheses formulated in Section 5.1.2. This procedure includes both quantitative and qualitative analyses. The quantitative data is the main source for testing the evaluation hypotheses, while the qualitative analysis provides a deeper insight into the quantitative results.

Quantitative Analysis Procedure

The precision and recall of the generated templates were calculated against reference templates. For each concept in scope a *mandatory* and an *optional* reference template were created. Mandatory reference templates represent the set of mandatory implementation steps, *i.e.*, the ones that are essential to the instantiation of a concept: if the step is removed, the concept does not work as expected. For example, without calling the method `createContextMenu()` (l. 29 in Figure 4.2) a context menu cannot be realized. For concepts that relate to the response to an event, such as Focus and Select, the mandatory steps are the ones that always occur as a result to the event. Optional reference templates additionally include steps that are not essential but that are relevant to the concept and were present in the sample applications. For instance, Context Menu's optional reference template includes calls to create and register separators.

Reference templates were carefully created to minimize threats to the validity of the results. For all concepts, reference templates were created using documentation found online (usually third-party articles or solutions posted in forums or both) and manually inspecting sample applications. In order to guarantee their correctness, reference templates were used in the creation of sample implementations. The determination of mandatory steps was mostly obvious with the help of framework documentation; dubious cases were verified by removing the step from the sample implementation and testing. Reference templates were then compared against the generated ones to identify optional features present in the sample applications. Each

non-mandatory step found in the generated template was examined and classified as optional, if it was relevant to the concept, or *irrelevant*, otherwise. If not clear, we were conservative and the step was considered irrelevant.

With respect to the above discussion, the calculation of precision and recall is based on counting the implementation steps contained in a template and comparing them against the reference templates. The considered implementation steps were superclass declarations, implement declarations (*i.e.*, declaring that a class implements an interface), method implementations (except `someMethod()`), method calls, and constructor calls. These steps are the main elements of a template. Call sequence and parameter passing patterns are considered supplementary information that makes the templates more readable. The calculation of precision and recall is based on determining three numbers:

- G: The number of all implementation steps in the generated template;
- M: The number of steps that are present in the reference template but missing in the generated template (*i.e.*, false negatives); and
- I: The number of steps that are incorrectly present in the generated template, but absent in the reference template (*i.e.*, false positives).

Precision (P) and *recall* (R) of a template are then calculated in the following way:

$$P = \frac{G - I}{G} \tag{5.1}$$

$$R = \frac{G - I}{G - I + M} \tag{5.2}$$

To reduce bias, the ranges of values possible for the precisions and recalls of generated templates were calculated using the mandatory and optional reference templates. The lower bounds of precisions and recalls were determined by using the mandatory reference templates in which all the optional steps in the generated templates were considered false positives. The upper bounds were calculated with the help of optional reference templates in which optional steps in the generated templates were not counted as false positives.

To analyze the influence of API trace slicing on the quality of the generated templates, all precision and recall calculations are also done for when full traces without slicing are used.

Qualitative Analysis Procedure

The generated templates were qualitatively analyzed to (1) gain qualitative insights into the false positives and false negatives identified in the quantitative analysis,

and (2) study the impacts of API trace slicing on the quality of templates and specify the situations in which conducting slicing is more useful than not applying it.

To perform the qualitative analysis, the generated templates were inspected and compared against the reference templates and the sources from which the reference templates were generated, *i.e.*, the documentation of the frameworks, third-party articles, actual example applications source code and their enclosing comments, and the solutions provided in developers forums.

5.4 Experiment Results

This section presents and discusses the results of quantitative and qualitative analyses of the generated templates conducted according to the procedures described in Section 5.3.6.

5.4.1 Quantitative Results

Table 5.2 illustrates the results of quantitative analysis conducted according to the procedure presented in Section 5.3.6. For the concepts with slicing, Table 5.2 also reflects the numbers obtained by using full traces without slicing to study the impacts of API trace slicing on the quantitative results. The final numbers are marked in bold, with precision ranging between 59% and 100% and recall ranging between 79% and 100% when optional reference templates are used. When mandatory reference templates are used (in parentheses), precision ranges between 12% and 100%, and recall ranges between 57% and 100%. The reason is that mandatory reference templates did not include any of the optional implementation steps, and hence, all those steps in the generated templates were counted as false positives. For example, for Toolbar Button, the code in the mandatory reference template would produce an empty button, *i.e.*, one without any icon, caption, and tool-tip text. When we compared the mandatory reference template with the generated one, the code in the generated template that corresponded to optional steps were counted as false positives. However, in real life, hardly any user is interested in the concept of an "empty toolbar button" as opposed to "toolbar button with some caption and icon". Therefore, we concentrate on the results for optional reference templates in the rest of this chapter.

The results for the *No Slicing* column, *i.e.*, using full traces, indicate that using the whole API trace is insufficient for getting quality templates because, although those templates contain a few false negatives, their size is larger than when slicing is used and they present more false positives. More specifically, slicing improved precision by eliminating between 20% and 80% of false positives, except for Context Menu, for which the sample applications were different enough to achieve 100% precision without slicing.

Table 5.2: The quantitative results of template extraction evaluation

Framework	Concept	Precision and Recall Results									
		No Slicing					With Slicing				
		G	I	M	P	R	G	I	M	P	R
Eclipse	Table Viewer	39	0 (23)	0 (0)	**100 (41)**	**100 (100)**	–	–	–	–	–
	Tree Viewer	45	0 (29)	1 (1)	**100 (36)**	**98 (94)**	–	–	–	–	–
	Navigate	40	10 (20)	0 (0)	75 (50)	100 (100)	38	8 (18)	0 (0)	**79 (53)**	**100 (100)**
	Focus*	4	0 (0)	0 (0)	**100 (100)**	**100 (100)**	–	–	–	–	–
JFace	Context Menu	15	0 (4)	1 (1)	100 (73)	94 (92)	15	0 (4)	1 (1)	**100 (73)**	**94 (92)**
	Toolbar Button	18	5 (14)	3 (3)	72 (22)	81 (57)	13	1 (9)	3 (3)	**92 (31)**	**80 (57)**
	Content Assist	46	27 (30)	1 (1)	41 (35)	95 (94)	32	13 (16)	1 (1)	**59 (50)**	**95 (94)**
GEF	Select*	7	0 (3)	0 (0)	**100 (57)**	**100 (100)**	–	–	–	–	–
	Figure*	83	25 (75)	0 (0)	70 (10)	100 (100)	68	10 (60)	0 (0)	**85 (12)**	**100 (100)**
	Connection*	91	26 (82)	0 (0)	71 (10)	100 (100)	76	10 (66)	0 (0)	**87 (13)**	**100 (100)**
	Title-bar Color*	–	–	–	–	–	–	–	–	–	–
Java 2D	Moving Shapes*	25	7 (16)	4 (4)	72 (36)	82 (69)	18	3 (9)	4 (4)	**83 (50)**	**79 (69)**
	Circle Drawing*	12	4 (9)	0 (0)	67 (25)	100 (100)	10	2 (7)	0 (0)	**80 (30)**	**100 (100)**
	Rounded Image*	–	–	–	–	–	–	–	–	–	–

These results quantitatively confirm the evaluation hypotheses that it is possible to extract implementation templates with high precision and recall from only two sample applications and/or execution scenarios implementing the desired concept in two different contexts. Moreover, these results confirm that API trace slicing can improve the precision of the templates.

5.4.2 Qualitative Results

This section presents the results of qualitative analysis of generated templates performed according to the procedure described in Section 5.3.6.

In general, false positives were more frequent than false negatives. False positives were due to similarities among the sample applications that extend beyond the concept of interest. For example, the remaining false positive for Toolbar Button was due to calls to `IShellProvider.getShell()`, which are frequently used

in Eclipse views. On the other hand, false negatives were mainly a result of (1) some difficulties in the current prototype implementation of FUDA, and/or (2) the existing issues in generating classes and their methods in template generation algorithms as described in Section 4.4.5. However, interestingly, there was not even a single false negative in the generated templates because of the API trace slicing. For instance, two of the false negatives for Moving Shapes were caused by the limitation of AspectJ which is used in FUDA Tracer. AspectJ cannot introduce code into `java.awt.*` packages or any other package belonging to the Java runtime library and therefore, the incoming calls were not traced. As an example of issues in template generation algorithms, for Tree Viewer, `getChildren()` was a framework-stipulated method that was implemented on the application side and there were no incoming calls to this method, except the outgoing calls. Therefore, this method was not implemented in the generated template.

Regarding the API trace slicing, it was particularly more useful when example applications shared more than the desired concept in common. For instance, for the GEF Figure, the example applications shared other concepts such as Palette and Toolbar as well, and therefore, performing slicing was highly beneficial. Slicing is also likely to be useful for traces generated using a single application, as such traces will likely have more common calls that are unrelated to the concept of interest. Moreover, using slicing and the current prototype implementation of FUDA, the user can see the effects of individual dependency types (*e.g.*, TT, TP, etc.) on the generated templates. Nevertheless, slicing is not required when (i) example applications share only the desired concept; in this case, concept trace slicing has no effect on the results compared to when the full traces are used; for instance, for the Context Menu, slicing had no effect; and (ii) the user is interested in the whole trace or just the events in the marked region; for instance, for the concept Table Viewer the whole traces were used, while for Focus, only the marked regions were used.

In the following, the qualitative analysis results are described for each concept individually:

- *Table Viewer and Tree Viewer:* For these two concepts, the concept defining question was how to implement a typical Eclipse table/tree viewer? Thus, the scenario involving view opening and closing spanned the entire view lifecycle. Consequently, API trace slicing was not needed.

  For Tree Viewer, as described earlier in this section, the `getChildren()` method was not implemented in the template and therefore was specified as a false negative.

- *Navigate:* Since both of the sample applications used for this concept had also the concept Context Menu in common, seven out of eight false positives for this concept were those creating the context menu for a view. Nevertheless, those false positives were easily identifiable by the user (almost all of them had the term `menuManager`).

- *Focus:* For this concept the user was interested only in the events that happened when she selected a view in the Eclipse environment. Therefore, it was not required to perform slicing, and only marked events were considered. This example shows that if the FUDA did not support marking, then full traces should have been taken into account which could have resulted in several false positives.

- *Context Menu:* The example applications for this concept had only the concept Context Menu in common. Therefore, performing slicing had no effect on the generated templates. There were no false positives and the only false negative was because of incorrectly identifying the framework packages. More specifically, `setMenu()` (l. 209 in Figure 4.5) was missing because it is in the Eclipse framework, not in JFace.

  This example indicates that if the sample applications are so different that they have only the desired concept in common, performing slicing may not be necessary. However, it is not always guaranteed to find such sample applications as the evaluations for other concepts shows.

- *Toolbar Button:* As described earlier, `IShellProvider.getShell()` was the only false positive which is widely used in Eclipse views. The false negatives were because of incorrectly identifying the framework packages, *i.e.*, they were in the Eclipse framework, not in JFace.

- *Content Assist:* Both of the sample applications were JFace-based text editors that had several concepts in common such as those that initialize and manipulate the text editors themselves. Again, the only false negative was because of incorrectly identifying the framework packages, *i.e.*, it was in the Eclipse framework, not in JFace.

  It is also worth mentioning that if the user knew the exact framework package name for Content Assist, *i.e.*, `org.eclipse.jface.text.contentassisst`, then the result would not have had any false positives.

- *Select:* The description is similar to what we had for the Focus.

- *Figure and Connection:* Slicing was particularly useful for these two concepts since all GEF editors use common parts such as palette and action bar. While steps related to the action bar were eliminated by slicing, some palette-related steps remained since palette was involved in all figure drawing scenarios.

- *Title-bar Color:* This concept was outside the scope of FUDA and therefore no templates were generated for it.

- *Moving Shapes:* As described before, two of the false negatives for this concept were due to the limitation of AspectJ that cannot introduce code into `java.awt.*` and therefore the incoming calls were not traced. Consequently, two method implementations were missed in the generated template. Nevertheless, this issue could have been addressed easily in this particular example

since those methods could have been determined by the interface of the class. The other two false negatives for this concept were due to different instructions that our sample applications used to change the location of a shape.

- *Circle Drawing:* Although there were no false negatives for this concept, the qualitative analysis revealed that one false negative was likely. The reason is that there are multiple ways of implementing circle drawing, *e.g.*, using `drawOval()` or `draw(new Ellipse)`, and the difference between these calls is not visible to the application user through its GUI. This concept and the Moving Shapes concept can be considered good examples of concepts that have exactly the same graphical representation, but can be implemented in different ways and therefore, introduce one potential threat to the applicability of FUDA. However, this issue can be addressed by applying data mining techniques and finding different alternatives in which a concept can be implemented.

- *Rounded Image:* This concept was outside the scope of FUDA and therefore no templates were generated for it.

5.5 Threats to Validity

This section discusses the potential threats that may impact the validity of the experiment results presented in the preceding section.

5.5.1 Internal Validity

The internal validity relates to the extent to which the design and analysis may have been compromised by the existence of confounding variables and other unexpected sources of bias [70]. In other words, potential threats in executing the steps of the experimental study are discussed.

The main threat to internal validity is incorrect reference templates which would impact the calculation of the data. This threat was minimized by (i) using three sources of knowledge for all concepts: manual inspection of sample applications, consulting existing documentation, and testing the implementation steps in sample implementations; (ii) both the author and another member of the Generative Software Development Lab independently checked in several iterations the correctness of all the reference templates and the values calculated for precisions and recalls; and (iii) reporting not only the values for the comparison with the optional reference templates, but also to the mandatory reference templates.

There might have been some problems with the prototype implementation of FUDA that could have led to wrong conclusions. To minimize this threat, the FUDA Tracer and a major part of FUDA Analyzer were debugged independently by another member of the Generative Software Development Lab. However, as

discussed in Section 5.4.2 for the concept Moving Shapes, there could be still some issues with the instrumentation of applications and frameworks because of the limitation of AspectJ which cannot introduce code into packages belonging to the Java runtime library.

5.5.2 External Validity

External validity relates to the extent to which the hypotheses capture the objectives of the research and the extent to which any conclusions can be generalized [70].

This experimental evaluation involves three inputs: frameworks, concepts, and sample applications. The way in which instances were selected for these variables directly affects the external validity of the results.

Frameworks. One potential threat to external validity is that the selection of frameworks for the evaluation might not have been representative of those used in real-world development. In particular, the selected frameworks are all GUI frameworks. This threat is addressed by selecting four popular frameworks that are widely used in practise. However, it might be still required to do experiments with other kinds of frameworks such as non-GUI frameworks; domain-specific frameworks that provide concepts such as data/control algorithms, transactions, and concurrency; and the frameworks that support reflection such as *Struts*[8] and *Spring*[9] which are quite common nowadays. Performing experimental study with these kinds of frameworks is left for future work.

Concepts. The selection of concepts for the evaluation might not have been representative of those used in real-world development. More specifically, they were all GUI concepts, self-contained, and very easy to delineate during invocation. This threat was addressed by including concepts from developer forums to answer real programming issues, and selecting concepts with different kinds of characteristics.

Sample Applications. The number of false positives and false negatives are highly dependent on how the given concept is implemented in the sample applications. Thus, the selection of sample applications for each concept directly influences the results. Based on this, one threat to external validity is that the way the sample applications selected for this experiment might not have been representative. This threat was minimized by following the same identification strategies that would be applied in practice, such as using the framework-packaged examples, searching the online open-source repositories, tips by others, and use a mix of them in the evaluation.

[8] `http://struts.apache.org/`.

[9] `http://www.springframework.org/`.

5.5.3 Construct Validity

The test of construct validity questions whether the theoretical constructs are interpreted and measured correctly [27]. In the case of this experiment, the main threat to construct validity is related to measuring the quality of generated templates by counting the number of false positives and false negatives. Additionally, our definition of false positives and false negatives can be a source of threat to construct validity. However, false positives and negatives are generally very subjective and depend on the user's definition of the concept. This threat was minimized by generating templates for typical functionalities of the concepts and reporting not only the values for the comparison with the optional reference templates, but also to the mandatory reference templates.

5.5.4 Reliability

Reliability of a study relates to demonstrating that the operations of a study can be repeated with the same results [143].

This chapter provided the setup of this experiment in detail, including the data collection and data analysis procedures. The sample applications used in this study are open-source. Moreover, the generated templates and reference templates are available online and can be obtained from [41]. In particular, the false positives and false negatives are highlighted in the generated templates. Consequently, it should be possible to replicate the experiment.

5.6 Summary

This chapter presented the results of an experimental study performed to evaluate the quality of the templates generated using the FUDA framework comprehension technique presented in Chapter 4. To this aim, FUDA was prototyped and this prototype implementation was used to perform the experiment for twelve concepts with different characteristics on top of four widely-used frameworks. The concept sample included both simple and complex ones. Six concepts corresponded to questions found at developer forums.

Both quantitative and qualitative analyses of the generated templates confirmed that FUDA can produce quality templates with high precision (59-100%) and recall (79-100%) from only two sample applications and/or execution scenarios. They also confirmed that API trace slicing can improve the results by eliminating between 20% and 80% of false positives, except for one concept, *i.e.*, the Context Menu, whose sample applications were different enough. The next chapter presents the results of an empirical study conducted to see whether the FUDA templates can be used in practise by software developers to perform real concept implementation tasks.

Chapter 6

Template Usage Evaluation

This chapter presents the results of a user experiment performed to analyze the usage of templates in the implementation of framework-provided concepts. This experiment was designed primarily to answer the research question of whether the implementation templates can serve as a substitute for framework documentation.

To answer the above research question, four concept implementation tasks on top of Eclipse were selected which varied in their complexity (two simple and two complex) and frequency (two frequent and two rare). Moreover, twelve highly skilled Java programmers were recruited to perform concept implementation tasks. These subjects were blocked into *experienced* and *moderate* groups based upon their previous experience with the Eclipse framework. Each subject was asked to implement one simple concept and one complex concept, each assisted by a different documentation aid (*i.e.*, implementation templates or framework documentation). The assignment of concept implementations was constrained by prior knowledge of the concepts. The experienced subjects were asked to implement the rare concepts and the moderate subjects were told to implement the frequent ones. Finally, both quantitative and qualitative analyses were conducted to compare the effectiveness of implementation templates and framework documentation in providing aid to subjects for developing framework-provided concepts.

The remainder of this chapter is organized as follows. Section 6.1 provides the details of this experiment's planning. Section 6.2 presents the experiment results followed by Section 6.3 that discusses the threats to the validity of these results. Finally, Section 6.4 summarizes this chapter.

6.1 Experiment Planning

This section presents how the experiment was designed and conducted.

6.1.1 Experiment Definition

The evaluation of the template extraction process presented in Chapter 5 showed that it is possible to automatically extract implementation templates with high precision and recall from only two sample applications. This experiment was designed to evaluate whether these templates are useful for application developers to realize framework-provided concepts. There are, of course, many possibilities here, given the wide variation in software development practices. However, in this experiment, the effectiveness of templates was compared with that of framework documentation in assisting the application developers. The rationale behind this is that if templates are as effective as framework documentation then they can serve as a substitute when no documentation is available.

With respect to the above discussion, the experiment attempted to answer the following research questions:

1. Are implementation templates as effective as framework documentation in aiding the development of framework-provided concepts, such that "effectiveness" is measured in terms of implementation time and resulting code correctness?

2. What is the influence of template quality and its usage strategies on the quality of resulting implementations? For example, if templates are simply pasted into target applications, the false positives could pollute implementations with undesired code, whereas false negatives could yield incomplete implementations.

6.1.2 Context Selection

The context selection is representative of situations where highly skilled Java programmers perform realistic concept implementation tasks for the first time on top of a non-trivial framework such as Eclipse.

More specifically, twelve highly skilled Java programmers were recruited. These subjects were a mixture of students and professionals, but mainly graduate students. Nevertheless, all the subjects except one had at least one year of industrial programming experience. The subjects were instructed to implement each assigned concept without interruptions at their discretion either at home or in the lab. Therefore, the results of this experiment can not be representative of implementation tasks in office settings in which phone calls, emails, and other distractions may interrupt the development.

Since all subjects had no previous familiarity with templates and had various degrees of experience with the Eclipse IDE and application frameworks, they were provided a personalized training as well as with written tutorials.

6.1.3 Hypothesis Formulation

The experiment has three independent variables with two factor levels each: *documentation aid* (framework documentation (D) and implementation template (T)), *concept complexity* (simple and complex) and *subject experience* (moderate and experienced). Moreover, it has two dependent variables:

- $\mathcal{T}$: Time to complete the assigned concept implementation task.
- $\mathcal{C}$: Functional correctness of the implementation code, measured as a factor with three levels:
  1. *Success*: If the implementation code behaves properly with respect to the task specification.
  2. *Buggy*: If the implementation code does not behave properly with respect to the task specification, but was considered a 'Success' by the subject.
  3. *Incomplete*: If the subject comes to the conclusion that she can not implement the assigned concept properly after a certain amount of time. Thus, she gives up the implementation and the code is incomplete.

Since we had no expectations as to which documentation aid is superior we formulated the following two-sided null hypotheses for the first research question presented in Section 6.1.1:

> $\mathbf{H_0}[\mathcal{T}]$: The time developers take to implement a framework-provided concept with the help of templates equals the time when implemented with the help of framework documentation. Formally, $\mathcal{T}(T) = \mathcal{T}(D)$.

> $\mathbf{H_0}[\mathcal{C}]$: The functional correctness of the implementation of a framework-provided concept when developed with the help of templates equals the functional correctness when implemented with the help of framework documentation. Formally, $\mathcal{C}(T) = \mathcal{C}(D)$.

Note that these hypotheses tell that implementation templates are as effective as the framework documentation in assisting the application developers to implement their desired concepts. However, they do not necessarily tell that templates are as effective as the framework documentation in understanding the framework API. A developer who has read a framework documentation might have a better understanding of the framework API than someone who has used a FUDA template to just implement a given concept.

For the second research question presented in Section 6.1.1, *i.e.*, the impacts of false positives and false negatives on the usage of templates and the quality of implementations, the subjects' concept implementations were qualitatively analyzed as will be explained in Section 6.2.2.

6.1.4 Experiment Design

The remainder of the subsections in this section present the design of this experiment. More specifically, they provide the choice of variables and the experiment procedure that was followed to test the hypotheses formulated in Section 6.1.3. These variables include:

- the choice of frameworks;
- the choice of concepts on top of the selected frameworks;
- the choice of target applications in which the subjects implemented their assigned concepts;
- the choice of sample applications to generate templates and give to the subjects as examples of actual implementations of the selected concepts;
- the choice of framework documentation; and
- the choice of subjects who performed the concept implementations.

In the following, first the choices of the above variables are described and then the procedures for conducting the experiment and analyzing the results are discussed.

6.1.5 Selection of Frameworks

Eclipse was chosen as the target framework because of the following reasons:

- Eclipse is a mature and complex framework that can be considered a good representative of modern object-oriented software frameworks.
- On top of Eclipse, it is possible to define concepts that can be implemented in a reasonable amount of time, which makes it suitable for performing this kind of experiments.
- Eclipse is quite popular these days and skilled Java programmers often have programming experience in the Eclipse IDE as well. Because of this, it is easier to find experienced subjects. As mentioned in Section 6.1.3, subjects' experience was one of the independent variables and the impact of experience on the results was of interest to this experiment.

6.1.6 Selection of Concepts

Five possible characteristics were mentioned for the framework-provided concepts in Section 5.3.2: in the scope of FUDA or not, requires slicing or not, atomic or composite, frequent or rare, and simple or complex. Since in this experiment the templates had been already generated, the only characteristics that might have influenced the results were the frequency and the complexity of the concepts. Frequency matters because experienced subjects typically have experience with the frequent concepts of the framework. In other words, experienced subjects can be distinguished from inexperienced ones with respect to their previous knowledge of frequent concepts. Complexity matters since it can determine the difficulty of a concept implementation task. Therefore, to study the impact of concepts' frequency and complexity on the results, two rare concepts (Content Assist and Navigate) and two frequent concepts (Context Menu and Table Viewer) were chosen for this experiment. Context Menu and Navigate were examples of simple concepts and Content Assist and Table Viewer were representative of complex ones. These concepts were also used in the evaluation of template extraction process as discussed in Section 5.3.2. It is also worth mentioning that one of the main reasons of selecting the concept Navigate for the purpose of this experiment was that it had the worst precision results (see Table 5.2 in Section 5.4.1) and evaluating the impact of several false positives on the quality of the results was of interest in this experiment.

The above concepts can be sorted based on their complexity in terms of number of lines of code: Navigate is the smallest one followed by Context Menu, Table Viewer and Content Assist, respectively. In this experiment the subjects were asked to implement a specific instance of these concepts as described in the following:

- *Context Menu.* Subjects were asked to implement a Context Menu in an Eclipse tree viewer with two *menu items*, labeled *Action 1* and *Action 2*, and are separated by a *separator*. The functionality of these menu items did not matter. Therefore, the subjects were told to, on their own discretion, either implement no functionality for these menu items or simply present a dialog-box showing that the menu item is selected.

- *Content Assist.* The content assistant feature of Eclipse text editors allows users to provide context sensitive content completion upon user request. For this concept, the subjects were asked to implement the functionality of the content assistant in a simple text editor in such a way that whenever the user enters the character dot (.) in the editor, a list of choices *Choice 1*−*Choice 5* should be opened.

- *Navigate.* The Eclipse framework provides a specific class with which developers can implement a simple web style navigation metaphor for a tree viewer. This metaphor supports the *Go Home*, *Go Back* and *Go Into* functions. For this concept, the subjects were told to create a toolbar for an Eclipse tree viewer and add these functions to it.

- *Table Viewer.* Implementation of this concept involves implementing an Eclipse table viewer consisting of three cells labeled *Cell 1*−*Cell 3* and a toolbar with one button. The functionality of this toolbar button did not matter: it could be empty or it could have presented a simple message on the developer's discretion.

6.1.7 Selection of Target Application

The target application for each concept was an incomplete Eclipse plug-in project in which the subjects were supposed to implement that concept. For each concept, the same target application was used for both T and D. To help developers focus on implementing their assigned concepts instead of wasting their time on investigating the source code of target applications, the size of these projects was kept minimal.

Particularly, the size of the target applications for the concepts Context Menu, Content Assist, Navigate, and Table Viewer were only 126, 10, 186, and 2 lines of code (LOC)[1] respectively. The target applications for the concepts Context Menu and Navigate were two simple Eclipse tree viewers that were generated automatically using Eclipse wizards. The target application for the concept Content Assist was a simple class that was extending the JFace's `TextEditor` class. Finally, the target application for the concept Table Viewer was an empty class.

6.1.8 Selection of Sample Applications

The same sample applications that were used to generate the concept implementation templates in Chapter 5 were used in this experiment as well. Consequently, the same templates that were generated in Chapter 5 were used again in this experiment. With respect to the precision and recall results presented in Table 5.2, these sample applications were good enough to generate quality templates with few false positives and false negatives. In other words, these sample applications provided actual implementations of the given concept in two different contexts.

These sample applications were given to the subjects in addition to the templates and documentation. The sizes of these sample applications varied between 1 KLOC (EditorList for Table Viewer) and 66 KLOC (Subclipse for Navigate).

6.1.9 Selection of Documentation

There were the following criteria in selecting the documentation for each concept:

- It should be the standard documentation of the framework.

[1] Excluding comments and blank lines.

- There should be explicitly one dedicated section in the documentation that describes how to realize the desired concept.
- It it preferable to contain either the complete template or the required code snippets to realize the given concept.

The documentation was selected in such a way that developers have the minimal effort for reading the text and that they can easily find the required implementation template or code snippets for realizing the concept. As it will be described later in this section, for all the concepts except Navigate, the selected documentation included either the complete template or the required code snippets such that some subjects were able to easily copy&paste the code from the documentation and perform some minor changes to it to implement their assigned concepts (see Section 6.2.2).

For this experiment, the documentation for a given concept was identified in Eclipse Help, Eclipse Corner Articles[2], or third-party Eclipse articles (web search). The documentation length varied between 5 pages (for concept Navigate) and 28 pages (for concept Content Assist). These documents were comprehensive, *i.e.*, they were providing some context information as well. For the concepts Navigate, Content Assist, and Table Viewer a second documentation with more background (or context) information about the desired concept was also provided. For the concept Context Menu, the provided documentation was already presenting enough background information and therefore there was no need to provide a second documentation. The documentation used in this experiment had a range of characteristics, which are described next:

- *Context Menu Documentation.* The provided documentation for this concept was the Eclipse Corner article on creating an Eclipse view[3]. This is the standard Eclipse documentation that describes how to create a basic Eclipse view. Moreover, this document includes a section that is dedicated to context menus and contains the full context menu code. The length of this article was 19 pages.
- *Content Assist Documentation.* The first documentation provided for this concept was the standard Eclipse help on the content assistant feature of JFace text editors[4]. This is a concise 3-page documentation that provides the required code snippets to implement a content assistant in a JFace text editor.

[2]Eclipse Corner Articles, `http://www.eclipse.org/articles/`.

[3]Creating an Eclipse View, `http://www.eclipse.org/articles/viewArticle/ViewArticle2.html`.

[4]Content Assist, `http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/editors_contentassist.htm`.

The second documentation was a non-standard third-party Eclipse article from the web[5]. This documentation is a comprehensive 28-page article that provides the details of implementing JFace text editors as well as the required code snippets for realizing the Content Assist.

- *Navigate Documentation.* The first documentation for this concept was the JavaDoc for the class `DrillDownAdapter`. The length of this documentation was 5 pages. At the beginning of this JavaDoc there is a paragraph that describes how to instantiate and use this class in order to create the navigation actions (*i.e.*, *Go Home*, *Go Back* and *Go Into*).

  The second documentation for this concept was the Eclipse Corner article on creating an Eclipse view as introduced earlier. The reason for providing this document was that the subjects were supposed to add the navigation actions to an Eclipse view's toolbar.

- *Table Viewer Documentation.* The first documentation for this concept was the Eclipse Corner article on creating an Eclipse view as introduced earlier. This article includes the full template for creating an Eclipse table viewer.

  The second documentation was a non-standard third-party Eclipse article from the web[6]. This documentation is a comprehensive 23-page article that provides in detail how to create an Eclipse table viewer. Nevertheless, this document also contains the full template for creating an Eclipse table viewer.

6.1.10 Selection of Subjects

Twelve highly skilled Java programmers, represented by $Subject_1$-$Subject_{12}$ or S_1-S_{12} for short, were recruited for the purpose of this experiment. All these subjects were recruited via direct contact throughout the experiment. These subjects were current and former members of the Generative Software Development (GSD) Lab[7] at the University of Waterloo as well as developers known to the author of this dissertation or his supervisor. The criteria for recruiting these subjects were proficiency with the Java programming language and general experience with object-oriented software frameworks. These subjects included one senior undergraduate student (S_{11}), nine graduate students (S_1-S_6, S_8-S_{10}), and two professionals (S_7 and S_{12}) from two different software companies. All subjects had between four and ten years of Java programming experience and all subjects except subject S_8 had at least one year of industrial programming experience in different software companies. Moreover, all the subjects were male.

[5]Building an Eclipse Text Editor with JFace Text, `http://www.realsolve.co.uk/site/tech/jface-text.php`.

[6]Using JFace Tables 3.3 API with Eclipse RCP - Tutorial, `http://www.vogella.de/articles/EclipseJFaceTable/article.html`.

[7]`http://gsd.uwaterloo.ca`.

Table 6.1: Subjects' background

	Subject	Occupation	Experience (Year)			Skill Level (1-5) (1= no skill, 5 = highly skilled)						Other Prog. Languages	Other Frameworks
			Industry	Prog.	Java Prog.	OO Ideas	Ideas of OO Fwks	Java Prog.	Eclipse IDE	Eclipse Fwk	JFace Fwk		
Experienced	S_1	Graduate Student	1	6	6	5	5	5	4	3	2	C++, C#, Python	EMF, .Net, Applets
	S_2	Graduate Student	2	5	5	5	5	5	4	4	4	C/C++, VB	SWT, Swing
	S_3	Graduate Student	2	15	5	5	5	5	5	5	3	Pascal, Basic, Standard ML	Applets, Eclipse JDT, Struts, Eclipse PDE
	S_4	Graduate Student	2	10	6	4	3	4	3	3	3	C++	J2EE EJB, Struts
	S_5	Graduate Student	2	17	7	4	4	4	3	3	2	C/C++, Assembly, Delphi	MFC, Swing, J2EE EJB
	S_6	Graduate Student	10	10	10	5	5	5	5	4	4	C++	OWL, MFC
Moderate	S_7	Professional	3	8	6	5	5	5	4	1	1	C/C++, C#, JScript	MFC, Hibernate, Spring
	S_8	Graduate Student	-	10	5	4	4	4	4	2	2	C/C++, Pascal	-
	S_9	Graduate Student	2	7	7	5	5	4	3	1	1	-	J2EE, DOM, XPath, ActiveBPEL
	S_{10}	Graduate Student	7	15	10	5	4	4	3	2	1	C/C++, Delphi	Servlets, Applets, OWL, HotDraw
	S_{11}	Undergrad. Student	1.5	4	4	4	4	5	5	1	1	Objective C	Cocoa, Servlets, JSP
	S_{12}	Professional	1	6	4	5	3	5	4	1	1	C/C++, C#, Pascal	Struts

Table 6.1 provides background data on the subjects that participated in the experiment. The subjects were blocked into two groups depending on their experience levels with the JFace and Eclipse frameworks:

- *Experienced Subjects:* The subjects in this group were proficient Java programmers who had previous experience with developing Eclipse plug-ins. Particularly, these subjects had implemented both frequent concepts before (Context Menu and Table Viewer), but not the rare ones (Navigate and Content Assist).
- *Moderate Subjects:* The subjects in this group had good Java programming experience, but had not previously implemented any of the four concepts in this experiment. In particular, they had no or little experience with the Eclipse and JFace frameworks.

6.1.11 Experiment Procedure

The following procedure was followed in this experiment as indicated in Figure 6.1:

Recruiting Subjects

In this step the potential subjects were contacted through email, phone, or in person. Each potential subject was given an introduction explaining generally the experiment in which he would be working. Furthermore, the subjects were questioned to see if they meet the experiment's criteria for recruiting subjects, *i.e.*, proficiency with the Java programming language and general experience with object-oriented software frameworks. In particular, the potential subjects were asked to see if they have any prior experience with the Eclipse framework.

Sending Subjects the Initial Package

In this step, an initial package was sent to subjects who met the experiment's criteria for recruiting subjects and agreed to participate in the experiment. This package contained the following documents:

- *An Overview Document.* A concise 1.5-page document providing the subjects with a brief introduction to FUDA, the purpose of the experiment, and the steps of the study that each subject would be going through.
- *A Background Questionnaire.* This questionnaire captured the subject's background and experience. As indicated in Table 6.1, some of the questions were: student or professional, industrial experience, familiarity with the ideas of object-orientation, knowledge of the ideas of object-oriented software frameworks, programming experience in general and Java programming experience

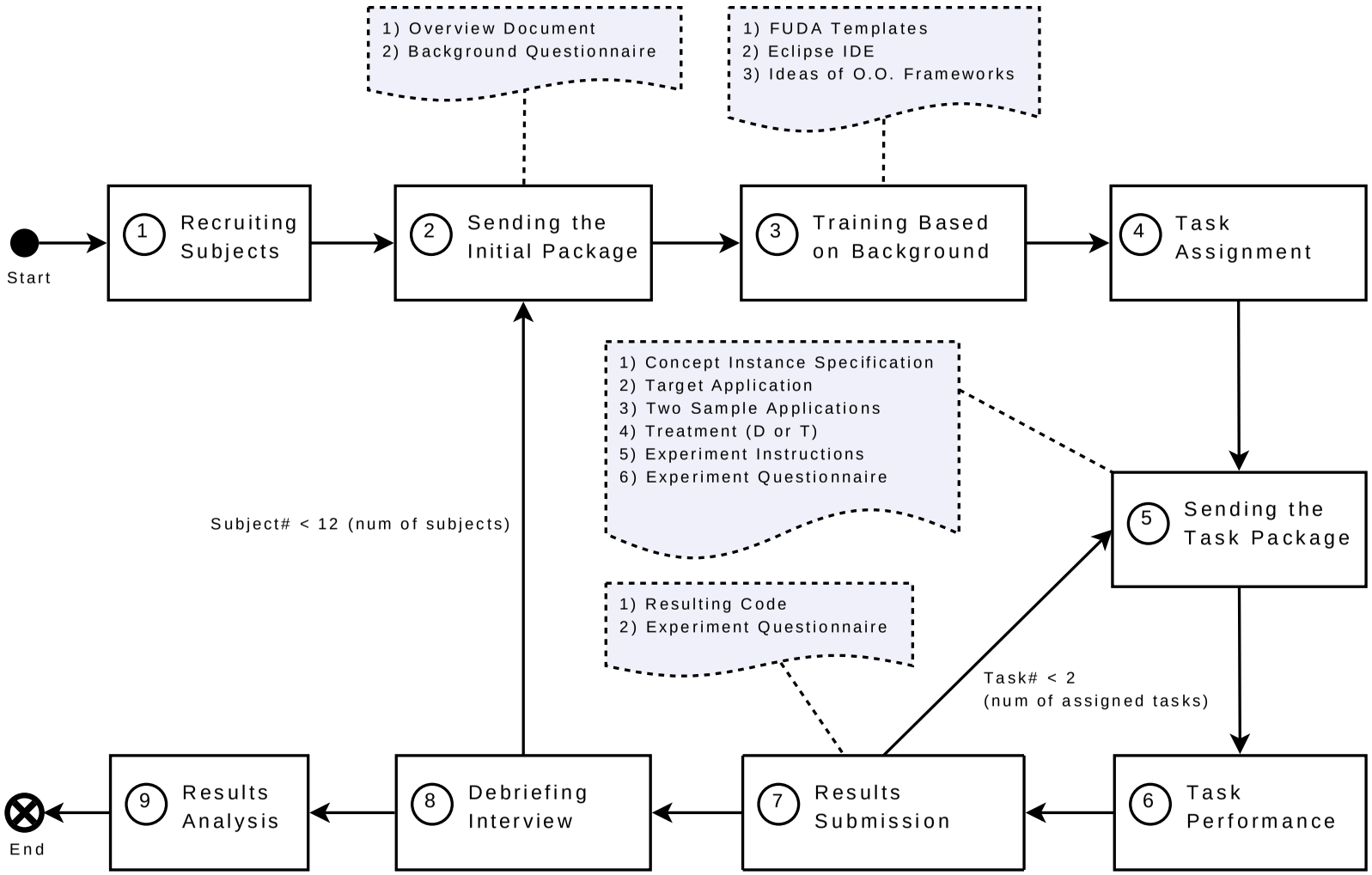


Figure 6.1: The procedure of template usage evaluation

in particular, knowledge of Eclipse IDE, whether or not he has implemented any of the four concepts of the study before, and what other frameworks and programming languages he is familiar with.

Training Subjects Depending on Their Backgrounds

Based on the subjects' answers to the questions in the background questionnaire, each subject was sent one to three different tutorials. These tutorials include:

- *The FUDA Templates.* This tutorial was a 3-page document describing the syntax and the semantics of FUDA templates as discussed in Chapter 4. This tutorial was sent to all subjects since it was the first time that they were going to use FUDA templates.
- *Introduction to Object-Oriented Software Frameworks.* This tutorial was a brief, 1-page document that was concisely introducing the ideas of object-oriented software frameworks such as framework API, framework-provided concepts, and mechanisms of using framework-provided services. This was done with the help of an example of Java applets. This document was sent to all subjects except the members of the GSD lab since they already had the required background. It is worth mentioning that although all the subjects had a general knowledge of software frameworks, it was noticed during the introduction sessions that they might have been using different terminologies. Therefore, this tutorial was sent to the subjects to make their terminologies homogeneous.
- *Introduction to the Eclipse IDE.* This tutorial was directed to the Eclipse IDE tasks that were required to be performed during the experiment, such as how to run an Eclipse plug-in under development. The length of this tutorial was 8 pages, but included several screenshots. This tutorial was sent to all moderate subjects since they had not implemented any Eclipse plug-ins prior to this experiment.

In addition to the above tutorials, each subject was given a personalized training session for about half an hour. During this session the subjects were questioned to become confident that they understood the ideas.

Concept Implementation Assignment

As mentioned earlier in Section 6.1.6, two frequent concepts (Context Menu and Table Viewer) and two rare concepts (Navigate and Content Assist) were selected for the purpose of this experiment. Furthermore, it was mentioned that the concepts Context Menu and Navigate were selected as examples of simple ones and the

Table 6.2: Sequence of performing concept implementation tasks

Subject		JFace Framework				Eclipse Framework			
		Context Menu		Content Assist		Navigate		Table Viewer	
		T	D	T	D	T	D	T	D
Experienced	S_1	-	-	①	-	-	②	-	-
	S_2	-	-	②	-	-	①	-	-
	S_3	-	-	②	-	-	①	-	-
	S_4	-	-	-	②	①	-	-	-
	S_5	-	-	-	①	②	-	-	-
	S_6	-	-	-	②	①	-	-	-
Moderate	S_7	-	②	-	-	-	-	①	-
	S_8	-	②	-	-	-	-	①	-
	S_9	-	①	-	-	-	-	②	-
	S_{10}	②	-	-	-	-	-	-	①
	S_{11}	②	-	-	-	-	-	-	①
	S_{12}	①	-	-	-	-	-	-	②

concepts Content Assist and Table Viewer were selected as examples of complex ones.

Based on the subjects' responses to the questions in the background questionnaire, two concept implementations were assigned to each subject. Assignment of two concept implementations to each subject helped to obtain more data points with fewer subjects. The assignment was constrained by prior knowledge of the concepts. In other words, the assignment made sure that the subjects had never implemented the given concept before. With respect to this, since all the experienced subjects had developed frequent concepts before performing this experiment, they were assigned to implement the rare concepts. Consequently, implementing the frequent concepts was assigned to moderate subjects. In this way, each subject was assigned one simple concept and one complex concept to implement. The assignment was random and balanced over the simple and complex concepts within each subject group (*i.e.*, moderate and experienced). The implementation sequence was also balanced within all subjects. In other words, six subjects first implemented the simple concepts and then implemented the complex ones, and six other subjects did the reverse.

Each subject was asked to use a template for one concept and documentation for the other concept. The subjects within each subject group were assigned randomly to documentation aids (T and D). The sequence of using the documentation aids was balanced both within each subject group and within all subjects. More specifically, in each subject group three subjects first used templates and then used documentation, and the other three subjects first used documentation and then used templates.

Table 6.2 illustrates the concept implementation assignments. In this table ① indicates the first concept that was implemented and ② identifies the second concept that was implemented by a subject. As can be seen in this table, each concept was implemented by three subjects using the templates and by three other subjects using the documentation. Therefore, in total we had twelve implementations assisted by the templates and twelve implementations aided by the documentation.

Sending Subjects the Task Package

After performing the concept implementation assignments, the next step was to send a task package for each of the assigned concepts to its corresponding developer. This task package constituted the following items:

- A *concept instance specification* presenting a precise description of the concept, a snapshot of what the implemented concept would look like, and a recommended implementation time. The concept instance specifications for all the concepts are provided in Appendix A. The recommended implementation times for the concepts Context Menu, Content Assist, Navigate, and Table Viewer were 60, 90, 45, and 90 minutes respectively. These times were determined based on the author's prior experience with developing these concepts and a dry run experiment done with one of the members of the GSD Lab.

- The *target application* in which the subject was supposed to implement the given concept as discussed earlier in Section 6.1.7.

- *Two sample applications* from which the concept implementation template had been generated as discussed earlier in Section 6.1.8.

- Either the *implementation template* (Section 6.1.8) or the *documentation* (Section 6.1.9) of the given concept.

- A document representing a set of *instructions* that the subject must have followed before and during the task performance. This document for the documentation aids T and D is provided in Appendix A. This document described to the subject what materials are included in the task package, how to set up the task performance environment (*e.g.*, importing the target project and sample applications into the Eclipse workspace), how to measure time, and a set of stepwise instructions to conduct the task performance. Furthermore, the subject was instructed about what he was allowed to do and what he was not allowed to do before and during the task performance.

  Before the task performance, the subject was allowed to read the tutorials and investigate the provided target application. The intent of the initial investigation phase was to prevent the case where a developer would try to perform the concept implementation task with almost no prior investigation of the target application. In other words, we wanted to measure the time that was actually spent on the concept implementation and not the target application investigation.

  Before the task performance, the subject was not allowed to read the provided template or documentation, and investigate the two sample applications that were actually implementing the concept.

During the task performance, the subject was allowed to use the provided template or documentation, investigate the provided sample applications source code, use the JavaDoc, and use the Eclipse Java editor code completion feature to complete the Java instructions. Note that JavaDoc documentation does not explain how to implement concepts, but only how to use a given framework-provided programming element such as an interface or a method.

During the task performance, the subject was not allowed to use the Eclipse wizards, the Eclipse help, the Internet to search for implementations of the concept, or any other source of information that may help to implement the concept except the provided sample applications and either the template or documentation. In other words, the only documentation aids the subject could use during the task performance were either the template or documentation, the two sample applications for the given concept, and the framework-provided JavaDoc documentation.

The subject was told to write down the time when he actually started implementing the concept, and write down the time when he either implemented the concept successfully or came into the conclusion that it was impossible to finish implementing the concept in a reasonable amount of time. Moreover, the subject was instructed to implement the given concept in one shot and without interruptions.

- A different *experiment questionnaire*, depending on whether the subject was supposed to implement the concept using the template or the documentation (see Appendix A). This questionnaire was used by the subject to provide us feedback on the experiment. The subject was instructed to fill this questionnaire only after the task performance. In particular, the subject was instructed to make short notes during the development, so that writing time does not impact the results.

  Some of the questions in this questionnaire for the subjects used templates were: how much time was spent on implementing the concept? was the concept successfully implemented or not? if not, why? how many and which example applications were used? was the template useful or not? was the format and the structure of the template okay? and, what kind of information were missing in the template?

  Some of the questions in this questionnaire for the subjects used documentation were: how much time was spent on implementing the concept? was the concept successfully implemented or not? If not, why? how many and which example applications were used? how many and which documentation was used? was the documentation useful? was the documentation concise enough? was the subject able to find all the required information in the provided documentation? and, was it easy to access the required information or not?

Task Performance

The task performance was done on subjects' discretion either at home or in the lab according to the instructions that were provided in the task package in the previous step.

Submitting the Results

After the task performance, the subjects were instructed to fill the experiment questionnaire provided in the task package and send it together with their concept implementation code to the author of this dissertation via email.

Debriefing Interview

After submitting the experiment questionnaire and the implementation code for both assigned concepts, each subject was asked to participate in a short informal debriefing interview.

Results Analysis

As the last step of this experiment's procedure, the subjects' concept implementation code was tested and inspected to see if the correct functionality was present as described in the concept instance specification in the task package. After collecting the results from all subjects, they were analyzed according to the analysis procedure provided in Section 6.1.13. The results of this analysis are presented in Section 6.2.

6.1.12 Instrumentation and Measurement

The instrumentation and measurement process was specified before the experiment began and determined precisely how the interaction with the subjects would be performed. Moreover, it outlined how the data would be collected when interacting with the subjects. As indicated in Figure 6.1, the data sources include:

- For every subject, an initial questionnaire capturing the subject's background and experience (see Table 6.1).
- For every task performance, an experiment questionnaire capturing the time spent on implementing the concept as well as the subject's feedback on the given template or documentation, and the sample applications.
- For every task performance, a copy of the concept implementation code.

- For every subject, a short informal debriefing interview done after the subject finished both of his assigned concept implementation tasks.

The resulting concept implementation code from each subject was tested and inspected to determine whether the concept was correctly implemented according to the concept instance specification or not. Full conformance to the concept instance specifications is important to be able to compare concept implementations in terms of development time and functional correctness. Given some subjects being slower than others, the time allocated for task performances was not fixed to (1) get as many concept implementations as possible, and (2) be able to observe differences in development times.

In addition to the background and experiment questionnaires, other materials for the experiment that were prepared in advance consisted of the experiment overview document, three tutorials (on implementation templates, object-oriented software frameworks, and how to use the Eclipse IDE), and, for each concept, the concept instance specification, the template and the documentation, the target application, the sample applications, and the instructions documents on how to conduct the experiment using the template or documentation.

6.1.13 Analysis Procedure

This section presents the procedure for analyzing the data collected during the experiment. This procedure constitutes both the quantitative and qualitative analyses. The quantitative data is the primary source for testing the evaluation hypotheses presented in Section 6.1.3, whereas the qualitative analysis attempts to gain a deeper understanding of the quantitative results and the work processes done by subjects.

Quantitative Analysis Procedure

The main aim of this quantitative analysis was to statistically test the hypothesis that the time to implement a framework-provided concept ($\mathcal{T}$) aided by implementation templates equals the implementation time when aided by framework documentation (see Section 6.1.3). For the dependent variable $\mathcal{C}$ (functional correctness of the implementation code), as it will be described later in Section 6.2.2, there were only two buggy implementations and, thus, it did not make sense to perform statistical analyses.

The quantitative assessment of the dependent variable $\mathcal{T}$ was performed through univariate statistical analyses, which were applied to each independent variable. For the independent variable documentation aid which was of particular interest, univariate analyses were performed to test the hypothesis individually for each concept, across simple and complex concepts, and across all concepts. Unpaired, two-sample t-tests [90] were applied. However, the t-test assumes that data are

drawn from a normally distributed population. To reduce potential threats to the validity of statistical conclusions resulting from violations of this assumption, the non-parametric *Wilcoxon* rank sum test [90] was also performed. This test does not rely on the assumption that data are drawn from a given probability distribution (*e.g.*, normal distribution).

The level of significance for the hypothesis tests was set to $\alpha = 0.05$. However, to allow for a stricter and more conservative interpretation of the results, p-values are calculated for all the tests. If the resulted p-value is greater than α, then the null hypothesis can not be rejected, and one can say that the difference between the D and T is not statistically significant with respect to that test.

Finally, it is also beneficial to know not only whether an experiment has a statistically significant effect but also the size of any observed effects. Thus, the effect size was computed both as the percentage difference between the two means (*%diff*), and using the Hedges' g [49], which is defined as the difference between two means divided by the pooled standard deviation for those means. In this experiment, since the difference in means between the T and the D groups was computed (T – D), a negative value of %diff and/or Hedges' g corresponded to the T documentation aid being more beneficial than the D documentation aid. To interpret the Hedges' g values, one suggestion is that $g = 0.2$ is indicative of a small effect size, $g = 0.5$ a medium effect size, and $g = 0.8$ a large effect size.

Qualitative Analysis Procedure

The main goals of this qualitative analysis were to (1) test the hypothesis that the functional correctness of a concept implemented with the help of templates equals the functional correctness of that concept when implemented with the help of framework documentation (see Section 6.1.3), and (2) understand the influence of false positives and false negatives on the usage of templates and on the quality of implementations (see Section 6.1.1).

As discussed earlier in Section 6.1.11, as the last step of this experiment's procedure, the subjects' concept implementation code was tested and inspected to see if it had the correct functionality and behavior as described in the concept instance specification in the task package. In this inspection, other code quality attributes such as the elegance and the structure of the code were immaterial since they make the evaluation subjective.

The experiment questionnaires were also qualitatively analyzed to extract and classify different kinds of information, such as the main difficulties in using templates, subjects' suggestions for improving the templates, for what purposes the example applications were mainly used, the quality of the provided documentation, and so on.

Finally, all the subjects participated in a short informal debriefing interview upon completion of their assigned concept implementation tasks. Debriefing inter-

Table 6.3: The subjects' concept implementation times in minutes

Subject		JFace Framework				Eclipse Framework			
		Context Menu		Content Assist		Navigate		Table Viewer	
		T	D	T	D	T	D	T	D
Experienced	S_1	-	-	50	-	-	25	-	-
	S_2	-	-	38	-	-	11	-	-
	S_3	-	-	48	-	-	10	-	-
	S_4	-	-	-	50	10	-	-	-
	S_5	-	-	-	41	10	-	-	-
	S_6	-	-	-	90	35	-	-	-
Moderate	S_7	-	10	-	-	-	-	30	-
	S_8	-	70	-	-	-	-	80	-
	S_9	-	27	-	-	-	-	50	-
	S_{10}	37	-	-	-	-	-	-	47
	S_{11}	18	-	-	-	-	-	-	38
	S_{12}	17	-	-	-	-	-	-	67

views helped us understand better the subjects' responses to experiment questionnaires, their style of working, whether they followed the experiment instructions, what were the main reasons of buggy implementations, and how they benefited from the provided documentation aids and sample applications (particularly their strategies of using the templates and documentation). The information collected during these interviews was used in the qualitative analysis of results (see Section 6.2.2)

6.2 Experiment Results

The results from the analysis procedures described in Section 6.1.13 are presented and discussed in this section. This section first provides the results of quantitative analysis followed by the results of qualitative analysis.

6.2.1 Quantitative Analysis Results

Table 6.3 indicates how many minutes each subject spent on implementing the given concept using the documentation aids T and D. Figure 6.2 also plots the time measured for each implementation as a function of the documentation aid and concept complexity. Bold labels identify experienced subjects; solid lines indicate the variance. Except subject S_8 who implemented the concept Context Menu assisted by the D in 70 minutes (> 60 minutes), the rest of the subjects were able to implement their assigned concepts within the recommended implementation times. Further investigation reveals that subjects S_6 and S_8 had the worst implementation times.

Table 6.4 presents the results of statistical analyses of concept implementation times indicated in Table 6.3 for each independent variable. This table presents the mean of the subjects' implementation times in the group (Mean), the standard deviation (Std Dev), the smallest (Min), the median (Med), the largest implementation time (Max), the effect size as a percentage difference between the two

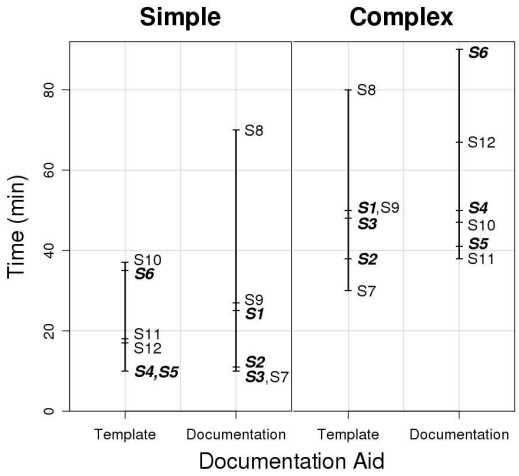


Figure 6.2: Plot of concept implementation times

means (%diff), the effect size using the Hedges' g, the p-value from the unpaired two-sample t-test, and the p-value from the Wilcoxon test.

The analysis of the statistical data presented in Table 6.4 reveals some interesting facts about the effect of documentation aids on the subjects' efficiency. The effect sizes %diff and Hedges' g, the variances, and the maximum data points indicate that across all the concepts together and across simple and complex concepts, the template group had slightly better results than the documentation group. In particular, the template group spent 13.0% less time in total, although this is not statistically significant according to the Hedges' g. If each concept is considered separately, it can be noted that for the concepts Context Menu and Content Assist the template group had better results and for the concepts Navigate and Table Viewer the documentation group had a little bit better results. The main reason for this is that for the first two concepts, the least efficient subjects (*i.e.*, S_6 and S_8) were using the documentation and therefore, the template group had better results. For the second two concepts, they were using the templates, and hence, the documentation group had a little bit better results. However, according to the Hedges' g, except for the Content Assist which the documentation aid T had a medium size effect, for the rest of the concepts the effect of the documentation aid was not significant.

If the medians are taken into account, it can be observed that except for the concept Table Viewer and across the complex concepts in which the documentation aid D was slightly better, for the rest of the concepts the documentation aid T was slightly better. However, the difference between the medians was typically negligible. Similarly, except for the concept Context Menu whose minimum implementation time was smaller for the documentation aid D, for the rest of the concepts the minimum implementation time was smaller for the documentation aid T.

Consideration of p-values shows that for all the tests, the computed p-values

Table 6.4: Statistical analysis of the subjects' concept implementation times

Independent Variable	Task	Factor Level	Mean	Std Dev	Min	Med	Max	%diff	Hedges' g	t-test (p-value)	Wilcoxon (p-value)
Documentation Aid	All Concepts	T	35.25	20.33	10.00	36.00	80.00	-13.0%	-0.22	0.5855	0.6851
		D	40.50	25.79	10.00	39.50	90.00				
	Simple Concepts	T	21.17	11.99	10.00	17.50	37.00	-17.0%	-0.22	0.6950	1.0000
		D	25.50	23.12	10.00	18.00	70.00				
	Complex Concepts	T	49.33	17.00	30.00	49.00	80.00	-11.1%	-0.31	0.5748	0.7466
		D	55.50	19.71	38.00	48.50	90.00				
	Context Menu	T	24.00	11.27	17.00	18.00	37.00	-32.7%	-0.40	0.5901	1.0000
		D	35.67	30.92	10.00	27.00	70.00				
	Content Assist	T	45.33	6.43	38.00	48.00	50.00	-24.9%	-0.63	0.4259	0.4000
		D	60.33	26.08	41.00	50.00	90.00				
	Navigate	T	18.33	14.43	10.00	10.00	35.00	19.6%	0.20	0.7747	1.0000
		D	15.33	8.39	10.00	11.00	25.00				
	Table Viewer	T	53.33	25.17	30.00	50.00	80.00	5.3%	0.10	0.8837	1.0000
		D	50.67	14.84	38.00	47.00	67.00				
Concept Complexity		Complex	52.42	17.84	30.00	49.00	90.00	124.6%	1.58	0.0006	0.0006
		Simple	23.33	17.71	10.00	17.50	70.00				
Subject Experience		Experienced	34.83	23.82	10.00	36.50	90.00	-14.9%	-0.25	0.5267	0.5431
		Moderate	40.92	22.49	10.00	37.50	80.00				

are greater than $\alpha = 0.05$. This means that based on the data points presented in Table 6.3, it is impossible to reject the null hypothesis $H_0[\mathcal{T}]$ with high significance for any of the tests. The fact that $H_0[\mathcal{T}]$ could not be rejected with high significance indicates that this experiment failed in providing evidence that templates and framework documentation are different in providing aid for developers to implement framework-provided concepts.

As the final point, it is worth mentioning that consideration of Table 6.4 for the independent variable concept complexity presents a different picture. The choice between a simple or a complex concept has an extremely significant ($p = 0.0006 \ll 0.05$) impact on development time. Complex concepts take consistently longer than simple concepts to implement (124.6% on average), regardless of documentation aid and experience. These trends can also be clearly verified by inspecting the diagram in Figure 6.2, where complexity deeply impacts time, while documentation aid does not.

6.2.2 Qualitative Analysis Results

This section presents the results of qualitative analysis conducted according to the procedure presented in Section 6.1.13. For this purpose, the concept implementations, the experiment questionnaires, and the information collected during the debriefing interviews were analyzed to gain qualitative insights into the impacts of using the templates or the documentation on the corresponding concept implementation task.

Following The Experiment Instructions

Debriefing interviews confirmed that all the subjects followed the experiment instructions provided in their task packages.

Code Inspection Results

This qualitative aspect of the evaluation involved the functional correctness of the concept implementations. Test and inspection of code for all concept implementations revealed two implementations that did not conform to the functional requirements. The first case was subject S_6's implementation of Content Assist aided by documentation where the requirements were not fulfilled. In particular, his implementation of Content Assist was being activated by pressing the default buttons (`Ctrl + Space`) in the text editor instead of pressing the `dot` ('.') button as specified in the concept instance specification. The second case was the implementation of Navigate by subject S_4 assisted by a FUDA template. In his implementation, an extra button was added to the view's toolbar besides the navigation buttons.

The remainder of the implementations satisfied the functional requirements of the corresponding concept instance specification. Since there were only two buggy

Table 6.5: Subjects' responses to questions on the experiment questionnaire for the documentation aid T

Subject		Concept	Overall Rank (1-5)[a]	#Samples Used	Content Useful?	Format Ok?
Experienced	S_1	Content Assist	5	2	✓	✓
	S_2		4	1	✓	-
	S_3		4	1	✓	✓
	S_4	Navigate	3	0	✓	✓
	S_5		4	1	✓	✓
	S_6		4	1	✓	-
Moderate	S_7	Table Viewer	4	2	✓	✓
	S_8		4	2	✓	✓
	S_9		2	2	✓	-
	S_{10}	Context Menu	3	2	✓	✓
	S_{11}		4	2	✓	✓
	S_{12}		5	1	✓	✓
Overall			Avg = 3.83	Avg = 1.42	#Checked = 12/12	#Checked = 9/12

[a] 1 = Not Useful, 5 = Highly Useful

implementations, each assisted by a different documentation aid, this attests the similarity of the templates and documentation in this respect.

Subjects' Comments for Templates

The analysis of experiment questionnaires and debriefing interviews provided insights into the subjects experience with implementation templates. Table 6.5 indicates the subjects' responses to questions on the experiment questionnaire and Table 6.6 summarizes and categorizes the specific comments regarding the templates. As these tables show, all subjects declared that templates were sources of useful information such that they ranked their usefulness 3.85 out of 5 on average. Some of the comments mentioned that templates were particularly useful to help focus on relevant classes and methods, and to indicate the portions of sample application code relevant to the task. This is reflected in a larger inspection of sample applications when using templates (1.42 on average) than when using documentation (0.75 on average, see Table 6.7). Subjects also observed that the syntactic resemblance between templates and Java code was important for understanding and copy&pasting templates into the target application.

Subjects also recommended aspects that could be improved. Many proposed features for a FUDA-specific graphical user interface (GUI), such as support for following the traceability links between templates and sample code, different *template views* that would show only the parts extracted from a single sample application, and decoration of templates with information from the sample applications. Some subjects remarked they lacked familiarity with the template in terms of syntax (such as the '`||`' notation) and of strategies that could be used to implement the concept using the information provided by the template. There were also some other comments, such as templates should point out which parts of them should be

Table 6.6: Summary of comments made by subjects about the templates

Category	Row	Comment	Subjects
Content	1	Templates were source of useful information	All Subjects
	2	The template was focused and concise	S_{11}
	3	The template helped me to quickly implement the concept without any prior experience with Eclipse plug-in development	S_7
	4	Templates do not tell anything about their contents	S_2, S_4, S_9, S_{10}
Java Syntax	5	Syntactic resemblance between templates and Java helped to understand the templates better	S_2, S_{11}-S_{12}
	6	Syntactic resemblance between templates and Java helped to not perform too much programming	S_1, S_4, S_8
	7	Some differences with the Java can be confusing at the beginning	S_6
	8	Defining all variables as global can be confusing	S_9
	9	It is not clear the difference between various method signatures separated by pipes (*i.e.*, `'||'`)	S_{11}
Reference to Samples	10	References to sample applications were very useful	S_1-S_3, S_7, S_{10}, S_{12}
	11	Templates helped to easily focus on relevant code snippets in the sample applications	S_1-S_2, S_{10}
GUI	12	Develop a GUI to easily jump from the templates to relevant code snippets in the example applications	S_2-S_3, S_9-S_{12}
	13	Develop a GUI in which the subjects can see the template of each sample application in isolation	S_{10}
	14	Inline the templates with the sample applications code to prevent redundant copy&paste actions	S_2
Usage Strategy	15	The strategy of using the templates is unclear	S_3-S_4, S_9
	16	Templates suffer from the lack of step-by-step flavor	S_2, S_9
	17	Templates have the issue that they encourage the "just-get-it-working" attitude without understanding completely what is going on	S_2, S_4
	18	The templates should point out which parts of them should be completed by the developer	S_8
Comments	19	The `"Unknown Order"` comment was frustrating	S_9
	20	Barely paid attention to annotations as `"Repeated!"`	S_{10}

completed by developers or the template should tell something about its content, which are very hard to be automated. Nonetheless, these observations suggest that the implementation of a user interface and more familiarity with the templates could improve the usability and potentially increase the efficiency of development with templates.

Subjects' Comments for Documentation

The analysis of experiment questionnaires and debriefing interviews provided insights into the subjects experience with the given documentation. Table 6.7 reflect the subjects' answers to questions that appeared on the experiment questionnaire. As this table shows, most of the subjects (ten out of twelve) declared that the given documentation contained enough information required for implementing the assigned concept such that they ranked its usefulness 3.67 out of 5 on average (note that this is very close to that of templates, *i.e.*, 3.83). Yet, subjects S_1 and S_4 noted that they were not able to find all the required information in the given documentation. In particular, subject S_1 complained about the lack of code snippets in the Navigate documentation. However, subject S_4 wrongly remarked that the Content Assist documentation did not advert that text editors must create their own `sourceViewer` class and he used sample applications to understand this fact.

The analysis of data in Table 6.7 reveals some interesting facts about the use of provided sample applications and documentation as well. As mentioned in the analysis of experiment questionnaires for templates, subjects inspected the sample applications much less often when using the documentation (0.75 on average) than when using the templates (1.42 on average). A more careful consideration shows that subjects mainly investigated the sample applications for complex concepts (Content Assist and Table Viewer) and not so much for the simple concepts (Context Menu and Navigate).

Most of the subjects only read the relevant parts of the first documentation. Interestingly, subjects S_6, S_8, and S_{12}, who read a large portion of the documentation, had the worst implementation times as well (see Table 6.3). All subjects except S_1 and S_5, who were provided two sets of documentation, also used the second one. S_1 told he did not read the second one because of his previous experience and S_5 said that he did not even notice that there was a second documentation as well. As Table 6.7 shows, subjects mainly skimmed the second documentation to get more background information.

Table 6.8 provides a summary of specific comments made by subjects about the given documentation and categorizes them into comments that are in favor of the given documentation and those that are not. For all the concepts, there were subjects who commented on useful code snippets and examples in the documentation (Rows 3-6). Subject S_{10}, who developed the Table Viewer concept, even mentioned that the example code and the text were nicely related through graphical arrows in the Eclipse Corner article on creating Eclipse views (Row 4). Moreover, subject

Table 6.7: Subjects' responses to questions on the experiment questionnaire for the documentation aid D

Subject		**Concept**	**Overall Rank (1-5)**[a]	**#Samples Used**	**Self-Contained**	**Easy Access**	**Concise**	**#Docs Used**	**Amount of Doc1 Read**	**Amount of Doc2 Read**
Experienced	S_1	Navigate	2	1	-	-	-	1/2	RP[b]	-
	S_2		5	0	✓	✓	✓	2/2	RP	RP
	S_3		4	0	✓	-	✓	2/2	RP	RP
	S_4	Content Assist	3	2	-	-	-	2/2	RP	RP
	S_5		3	2	✓	-	-	1/2	RP	-
	S_6		4	1	✓	✓	✓	2/2	BP[c]	RP
Moderate	S_7	Context Menu	5	0	✓	✓	✓	1/1	RP	-
	S_8		4	0	✓	-	✓	1/1	BP	-
	S_9		3	0	✓	-	✓	1/1	RP	-
	S_{10}	Table Viewer	4	0	✓	✓	✓	2/2	RP	RP
	S_{11}		3	1	✓	-	-	2/2	RP	RP
	S_{12}		4	2	✓	-	-	2/2	BP	RP
Overall			Avg = 3.67	Avg = 0.75	#Checked = 10/12	#Checked = 4/12	#Checked = 7/12	-	-	-

[a] 1 = Not Useful, 5 = Highly Useful
[b] RP: Only Relevant Parts
[c] BP: A Big Portion

Table 6.8: Summary of comments made by subjects about the documentation

Category	**Row**	**Comment**	**Subjects**
Comments in Favor	1	The documentation was adequate for implementing the concept	S_2-S_3, S_4-S_{12}
	2	The documentation was useful in explaining the purpose and the functionality of the concept	S_1
	3	Had good examples	S_2, S_4, S_8-S_{10}
	4	The text and the code snippets were nicely related through arrows	S_{10}
	5	It was impossible to realize the concept without the examples	S_9
	6	It was easily possible to realize the concept by just reading the section dedicated to the concept	S_7
	7	Good correlation to the provided sample applications	S_4-S_5
Comments Against	8	Need to filter a lot of unrelated material to figure out the concept	S_1, S_4-S_5, S_8, S_{10}-S_{11}
	9	A long document	S_8
	10	Had to read a big portion of the document	S_8
	11	Need to do hunting in the documentation to understand all the required information	S_9
	12	The documentation assumed it is being read from start to finish	S_9
	13	The text and the code snippets did not link well	S_4
	14	The document was in the form of a walkthrough	S_5
	15	The documentation was incomplete	S_1, S_3
	16	The documentation does not say how the concept should be typically implemented	S_1, S_3
	17	Had no sense whether or not the concept had been implemented correctly	S_1, S_3

S_9 alluded that without those code snippets he would not have been able to realize the Context Menu concept (Row 5). Finally, two of the subjects (S_4 and S_5) observed the strong correlation between the documentation and the given sample applications for the concept Content Assist (Row 7). As will be discussed later, both of these subjects used the documentation mainly as the starting point for investigating the sample applications.

In addition to the above comments in favor of the given documentation, the subjects also commented on several perceived shortcomings. However, it is also worth mentioning that different subjects might have had different views on the same documentation. For instance, although subject S_7 said it was easy to realize the Context Menu by just reading the section dedicated to this concept, subject S_8 mentioned that he needed to read a big portion of the documentation for the same task.

For all the concepts there were subjects who complained that they had been required to filter out a lot of irrelevant material to realize the concept (Row 8). Because of this, only four out of twelve subjects declared easy access to information in the given documentation and five subjects believed that the documentation was not concise enough (see Table 6.7). In particular, for the Context Menu concept, subject S_8 said that the documentation was long and he did read a big portion of the documentation (Rows 9-10). On the other hand, subject S_9 jumped directly to the section dedicated to the context menus. Nevertheless, he later was required to go back in the documentation to understand the code completely (Rows 11-12).

For the Content Assist concept (Rows 13-14), although S_4 alluded that there were good examples in the documentation, he complained that those examples were not linked well to the text and S_5 said that the documentation had the form of a walk-through.

For the Navigate concept (Rows 15-17), two out of three subjects had difficulty with the lack of code snippets in the documentation, which led them to consider the documentation incomplete and not be confident on the correctness of their implementation. This particular example shows that although there was a good description of the Navigate concept in the documentation to realize it, developers prefer to see actual code snippets rather than plain English text, even for simple concepts whose implementations are only a few lines of code.

Template Usage Strategies

By analyzing the experiment questionnaires and performing the debriefing interviews, it was noticed that the subjects basically followed five different strategies to use the given implementation template (TS stands for *Template usage Strategy* in the following):

TS_1. Copy&pasting the code from the template into the target application; then, investigating the sample applications to refine that code.

Table 6.9: The subjects' strategies of using the FUDA templates

Subject		Concept	Followed Template Usage Strategy
Experienced	S_1	Content Assist	TS_1
	S_2		TS_4
	S_3		TS_3
	S_4	Navigate	TS_2
	S_5		TS_5
	S_6		TS_5
Moderate	S_7	Table Viewer	TS_1
	S_8		TS_1
	S_9		TS_5
	S_{10}	Context Menu	TS_4
	S_{11}		TS_3
	S_{12}		TS_3

Table 6.10: The impact of false positives and false negatives on the concept implementation code

Concept	I	M	Effect
Context Menu	0	1	The false negative caused the implementation to fail.
Content Assist	13	1	The false positives caused a null pointer exception at runtime. The false negative caused the content assistant to not show up.
Navigate	8	0	The false positives created an extra button in the view's toolbar.
Table Viewer	0	0	–

TS_2. Copy&pasting the code blindly from the template into the target application without investigating the sample applications.

TS_3. Copy&pasting the code blindly from the template; after encountering some issues at runtime, inspecting the template code and sample applications to find out the reasons of those issues.

TS_4. Using the template just as an entry to sample applications; then, copy&pasting the code snippets from the sample applications into the target application.

TS_5. Using the template just as an entry to sample applications; then, investigating the sample applications to learn their code; next, writing the code in the target application by the subject himself.

Table 6.9 illustrates each subject followed which one of the above template usage strategies.

Impact of False Positives and False Negatives. Table 6.10 indicates the impact of false positives and false negatives on each concept's implementation code. As it shows, the concepts either did not work properly and it was visible at runtime (the Context Menu did not show up, the runtime exception for the Content Assist, or the extra button for the Navigate), or there were some compile errors in the code (for the Table Viewer). Following describes how each subject tackled the issue of false positives and false negatives in the templates:

- The template extracted for the concept Context Menu contained a false negative which prevented the context menu from showing up at runtime by simply

following the template. The subjects S_{11} and S_{12} were able to address this issue by looking at the sample applications and noticing that one instruction (`setMenu()`) was missed in the template. Since subject S_{10} used the template only as an entry to sample applications, he did not encounter this issue.

- The template for the concept Content Assist included several false positives. However, these false positives did not result in major problems for the subjects S_1 and S_2 since S_1 refined the template code using the sample applications before actually executing the target application, and S_2 used the template as an entry to sample applications and got the concept implementation code from them. However, subject S_3 got a runtime null pointer exception by simply using the template code. To address this problem, he inspected the template code and noticed that one of the overridden methods in the template was not required. All the subjects were able to detect the false negative in the template using the sample applications.

- As mentioned earlier, subject S_4 had a buggy implementation of Navigate using the template. As Table 6.9 shows, he blindly copy&pasted the code from the template into the target application without investigating the sample applications. This polluted the target application with false positives (see Table 6.10), *i.e.*, code that was not related to the task. Since the developer, in this case, did not inspect the sample applications, this code remained in the target application, and caused an unwanted button in the view's toolbar. However, although this button was visible in the toolbar, the subject did not try to solve the issue. Subjects S_5 and S_6 did not have difficulty with using the template for this concept since they used the template as the starting point to learn how the sample applications implement the Navigate concept.

- The template for the concept Table Viewer contained neither false positives nor false negatives. None of the subjects reported any difficulty with using the Table Viewer template.

With respect to the above discussion, it can be concluded that there were mainly two use cases for templates:

1. Copy&pasting the concept implementation code from the templates into the target applications. In this case, the subject must be careful with false positives and false negatives.

2. Using the templates as entries to sample applications. In this way, templates can be considered as the output of feature location that provides a summary of relevant program instructions as well.

These observations suggest that templates must be used together with sample applications since they help to understand what is missing and to detect unneeded code. Interestingly, the only subject (S_4) who did not refer to sample applications (see Table 6.5) had a buggy implementation.

Documentation Usage Strategies

Qualitative analysis of experiment questionnaires and debriefing interviews revealed that the subjects used a variety of strategies to benefit from the provided documentation (DS stands for *Documentation usage Strategy* in the following):

DS_1. First, reading a big portion of the text to get the background information; copy&pasting the code template from the documentation into the target application and fixing it.

DS_2. Jumping directly to the code snippet in the documentation; copy&pasting the code template from the documentation into the target application; reading the text to understand the code and modifying it if needed.

DS_3. Reading the relevant parts of the documentation to understand the involved classes; jumping into the sample applications using those classes; copy&pasting the code snippets from the sample applications into the target application and amending them if needed.

DS_4. Reading the relevant parts of the documentation to understand the involved classes; jumping into the sample applications using those classes; copy&pasting the code snippets from the sample applications into the target application; reading again the documentation to understand better those code snippets and performing the necessary changes.

DS_5. Reading the relevant parts of the documentation; copy&pasting the code snippets from the documentation into the target application; comparing the implemented code against the sample applications to confirm its correctness.

DS_6. Reading the relevant parts of the documentation; copy&pasting the code snippets from the documentation into the target application followed by some implementations by the subject himself.

DS_7. Reading the relevant parts of the documentation; next, implementing the concept by the subject himself from scratch.

Table 6.9 indicates each subject followed which one of the documentation usage strategies. As can be seen in this table, subjects used the documentation in more diverse ways compared to the templates. However, it could be understood that the sample applications were mainly used either to get the concept implementation code snippets or to confirm that the concept was implemented properly.

6.2.3 Discussion

The quantitative and the qualitative analyses confirmed that the null hypotheses presented in Section 6.1.3 can not be rejected based on this experiment's data. In

Table 6.11: The subjects' strategies of using the documentation

Subject		Concept	Followed Documentation Usage Strategy
Experienced	S_1	Navigate	DS_3
	S_2		DS_7
	S_3		DS_6
	S_4	Content Assist	DS_4
	S_5		DS_3
	S_6		DS_5
Moderate	S_7	Context Menu	DS_2
	S_8		DS_1
	S_9		DS_2
	S_{10}	Table Viewer	DS_2
	S_{11}		DS_4
	S_{12}		DS_5

other words, they failed in providing evidence that implementation templates and framework documentation are different in assisting the application developers to realize their desired framework-provided concepts. However, in these analyses the following points should be taken into account:

- All the subjects had no prior experience with using templates before this experiment; thus, a certain learning curve can be expected. This was also observed by a number of subjects such as S_6, S_8 and S_{10}. They mentioned that it took them a while to get used to the templates and if they were supposed to implement another concept with the help of FUDA templates it would have been easier for them.

- There was no FUDA-specific GUI, specially one for following the traceability links between templates and sample applications code. Because of this, the subjects needed to manually search the sample applications for traceability links which was a cumbersome task. As it was also confirmed by a number of subjects such as S_{10} and S_{11}, if there were a nice GUI for this purpose, the subjects might have had a better efficiency.

Given the above circumstances, this experiment indicated that FUDA templates are comparable to good framework documentation that provide complete descriptions as well as some coding examples of desired concepts in such a way that some subjects only copy&pasted the code from the documentation and performed some minor modifications to it (*e.g.*, Subject S_7 for the concept Context Menu). However, for many frameworks and concepts, there is no documentation. In this case, the FUDA technique would be even more useful.

Documents also have the issue of being typically written in only a few languages (*e.g.*, in English). Therefore, developers, who do not know the language in which the document is written, may not be able to understand that document as well. However, templates are in Java pseudo-code, which can typically be understood by many software developers. Moreover, documents are written by humans and may become outdated, while the FUDA templates can be generated automatically.

6.3 Threats to Validity

There are several factors that may potentially affect the validity of the results of this experiment. This section provides a description of these factors.

6.3.1 Internal Validity

The main threat to internal validity concerns the distribution of subjects over the concept implementation tasks. This threat was minimized by blocking subjects according to experience and randomizing the remaining distribution. Moreover, the subjects were constrained by their prior knowledge of concept implementation tasks.

Another possible source of threat to internal validity relates to selection of subjects using convenience sampling instead of random sampling. This may introduce some bias in selecting the subjects. This threat was mitigated by contacting only the professional Java programmers who had no prior knowledge about the assigned concepts. Moreover, the subjects were blocked by experience and randomly assigned to documentation aids within each block.

Since all the subjects performed two concept implementations, this may introduce learning effects from one task to the next. This threat was minimized by balancing the sequence of using the documentation aids and balancing the sequence of concept complexity. In particular, half of the subjects first used the templates for the first concept implementation, and then used the documentation for the second concept implementation, and the other half of the subjects did the reverse. Similarly, half of the subjects first implemented the simple concepts and then implemented the complex concepts, and the other half of the subjects did the reverse.

One other source of interference could be the subjects' prior knowledge and proficiency with the development environment. To reduce this threat, the subjects were blocked based on their experience, none of the subjects had previous knowledge of FUDA templates, the moderate subjects were provided basic tutorials of Eclipse and object-oriented frameworks, it was banned the use of powerful features of Eclipse such as the debugger, and the entire study was scripted.

Finally, there is the threat of using wrong statistical analysis methods. This threat was minimized by consulting four statisticians from the Department of Statistics and Actuarial Science at the University of Waterloo.

6.3.2 External Validity

The principal threat to external validity refers to the generalization from students to professionals. This threat was mitigated using a combination of mostly graduate students (9/12), two professionals, and one senior undergraduate student. All the

subjects were highly skilled programmers with at least four years of Java programming experience. Moreover, all the subjects except one had at least one year of industry experience. In particular, the only undergraduate student who participated in this study, had 1.5 years of industry experience.

Selection of subjects through convenience sampling instead of random sampling can be a potential threat to external validity as well. However, the participants in this experiment had different levels of experience with the Eclipse framework, they were from different universities, they had different nationalities, and they had industry experience in different companies.

The size and the complexity of concept instances used in this experiment is another source of threat to external validity. However, this is not actually a threat since the experiment intentionally targeted the concept instances that could be implemented in a short period of time as described in Section 6.1.6. Nevertheless, two simple concepts and two complex concepts were used in this study.

All the concepts selected for this study were GUI concepts on top of the Eclipse framework. Hence, the results cannot be necessarily generalized to other types of concepts and frameworks. However, as mentioned in Section 6.1.5, Eclipse is a mature, complex, mainly GUI framework that can be considered as a representative of many of the modern object-oriented software frameworks.

Another potential limit to the generalizability of this experiment relates to generalization from the home or laboratory settings to real settings. This threat was mitigated by using a realistic context with state-of-the-art technologies (*e.g.*, Eclipse IDE).

Finally, one possible constraint to the generalizability of the results comes from the fact that the results of this experiment are based on using twelve subjects. We might be able to gain more confidence in the results by recruiting more subjects from a larger cross-section of backgrounds.

6.3.3 Construct Validity

In the case of this experiment, the main threats to construct validity are related to:

1. To compare the effectiveness of T and D in providing aid to developers, it must have been made sure that the subjects have actually used the given documentation aids.

2. Since the effectiveness of templates are compared with that of the documentation in assisting the application developers, the definition of documentation must be clear.

3. Because the effectiveness of documentation aids in assisting the subjects are compared, the measurement of effectiveness should be precisely specified.

Regarding the use of documentation aids, all subjects in their experiment questionnaires reported on the usefulness of the provided documentation aids and provided some comments about them. This is an indication that all subjects used the given documentation aids.

The definition of documentation used in this experiment is provided in Section 6.1.9. As described in that section, the definition of documentation sought to maximize its familiarity and conciseness by selecting standard documents dedicated to the concept at hand.

The measurement of effectiveness was done by the implementation time and the functional correctness of the concept. It is clear that different notions could be used, such as code quality in terms of its modularity, that could affect the results. However, in order to reduce the subjectivity of the assessment, source code inspection and testing was done only to make sure that the implemented concept had the correct behavior as described in the concept instance specification. It is also worth mentioning that regarding the measurement of time, the subjects were trusted that they would report correct times.

6.3.4 Reliability

This chapter documented the methodology of this experiment, including the data collection and data analysis procedures. The complete experimental material as well as the subjects' background questionnaires and experiment questionnaires can be obtained from [41]. Appendix A also presents the experimental material used in this study. The target applications and the sample applications used in this study are open-source. Moreover, the concept implementation templates and documentation are available online. As a result, it should be possible to replicate the experiment.

6.4 Summary

This chapter presented the results of a user experiment with twelve subjects conducted to compare the effectiveness of implementation templates with that of framework documentation in providing aid for developers to implement framework-provided concepts. For the studied sample, no statistically significant difference between using templates and documentation in terms of implementation time and number of introduced bugs could be detected. More specifically, for the studied sample, the choice of templates vs. documentation had much less impact on development time than the concept complexity. As a result, this experiment suggests that templates can be considered as a replacement for documentation where no or little documentation is available. This situation could become even better with the implementation of a FUDA-specific user interface and subjects' more familiarity with the templates.

Finally, the qualitative analysis of results indicated that the templates should be used together with the sample applications from which they were extracted rather than just by looking at the templates alone in order to understand what is missing in the templates (false negatives) and to detect unneeded code (false positives).

Chapter 7

Conclusions

This chapter provides the discussion in Section 7.1, features the summary of this dissertation in Section 7.2, and suggests a number of future research directions in Section 7.3.

7.1 Discussion

This section talks about a number of items that require further elaboration: strengths and weaknesses of the FUDA framework comprehension technique, considerations that should be taken into account in designing the concept-invoking scenarios, some points about the API trace slicing, and the developers' option to incrementally comprehend the concept implementations using the current prototype implementation of FUDA.

7.1.1 Strengths and Weaknesses

The results of template extraction evaluation presented in Chapter 5 indicate that FUDA can retrieve concept implementation templates with relatively high precision and recall from only two sample applications. Furthermore, the approach is highly automated. The processing of the traces is fully automatic and the instrumentation does not impose significant overhead on the application execution since only the API interaction rather than full traces are recorded. Given a set of applications and scenarios, the amount of time needed to retrieve templates is mainly determined by the amount it takes to execute the scenarios on the applications. Furthermore, dynamic analysis detects the API elements that are actually being invoked. This is important since frameworks typically use polymorphism and reflection, which can render static analysis less precise. Finally, the results of the template usage evaluation discussed in Chapter 6 revealed that FUDA templates can be used instead of framework documentation when no documentation is available.

Nevertheless, the approach has some potential drawbacks as well. Most importantly, it relies on the ability to find appropriate sample applications. The quality of the results may depend on the selection of the applications and concept invocation scenarios. In particular, creating the scenarios might require careful design to isolate the API instructions of interest in the context of composite concepts. Second, all dynamic approaches are dependent on the input data and generalizing from this data might not be safe. In that sense, FUDA will retrieve the set of API instructions that are invoked in each execution, but may fail to retrieve optional API instructions. Both issues are discussed further in the next subsection. Finally, dynamic approaches require the setup of the runtime environment, which might not be easy in some situations. Therefore, being able to retrieve useful concept implementation templates from only few application executions is particularly important.

7.1.2 Scenario Design Considerations

The nature of the concept and the ways in which it is implemented by the applications can influence the results. Ideally, the concept is atomic, its invocation is easily delimitable (for marking), and the sample applications have only this concept in common. In this case, FUDA will yield best results. In general, concepts are composites of other concepts, the invocation of a concept might not be easily demarcated, and the sample applications may have several concepts in common. For a composite concept, developers should select applications that vary those of its components that should be eliminated. If the concept of interest is part of a composite concept, developers should be able to demarcate the boundaries of the concept execution. For example, most Eclipse views implementing context menus also implement *actions*, *toolbars* and other Eclipse concepts, but the menu execution can be marked. However, in some cases even marking cannot isolate the concept of interest. For example, for the concept GEF Figure, the palette was involved in all figure drawing scenarios, and hence, the extracted template contained some palette-related steps (see Section 5.4.2). Nonetheless, developers could address such issues by following the traceability links in the template and studying the actual sample application code.

7.1.3 API Trace Slicing

It is worth noting that the API trace slicing presented in Section 4.4.4 is significantly different from traditional dynamic program slicing techniques discussed in Section 3.6. First, while program slicing is defined in terms of data and control dependencies, API trace slicing is defined in terms of object-relatedness between API calls, which is induced by the use of common objects as targets, parameters, or return values. Furthermore, while traditional dynamic slicing is defined with respect to a trace containing all program instructions that were executed, API trace slicing operates on API interaction traces.

Object-relatedness is motivated by common API usage patterns. For example, two method invocations sharing the same target object could be related by the fact that one call initializes the target for the second call, or the second call cleans up the object that was used in the first one. Similarly, a call that returns an object which is later used as a target or parameter in a subsequent call may be an invocation to a factory method. Clearly, object-relatedness as defined in this dissertation may lead both to false positives and false negatives. For example, false positives commonly occur if the same object is used in two calls for unrelated uses. False negatives can happen if invocations are related by side effects, such as accessing some objects in some framework registry. Nevertheless, as shown in the template extraction evaluations, many of the false positives would be eliminated in the common fact extraction stage, and situations leading to false negatives are typically uncommon in the first place.

7.1.4 Incremental Analysis

To enable the application developers to incrementally comprehend how their concepts of interest are implemented in example applications, in the current prototype implementation of FUDA Analyzer (see Section 5.2.2), it is possible to select various API trace slicing and template generation options. For example, one can enable or disable different object dependency types to be considered in the API trace slicing. In this way, the developer can build her knowledge about the implementation of a given concept incrementally and see the impact of individual dependency types or a combination of them on the results. Moreover, in the current prototype implementation of FUDA Tracer (see Section 5.2.1), the developer can define different filters for the events to be recorded. Using these filters, she can narrow down her focus to the events of her interest only. For example, if she is interested only in those events in which the method `createPartControl()` is involved, she can simply define this method as the filter.

7.2 Summary

Although object-oriented application frameworks are one of the most effective reuse technologies available today, many of them are difficult to use because of their large and complex APIs and often incomplete user documentation. To cope with this problem, application developers often use existing framework applications as a guide to comprehend how to implement a desired framework-provided concept. While existing applications contain valuable examples of concept implementation steps, locating them in the application code is often challenging. To address this difficulty, this dissertation introduced the notion of concept implementation templates and devised a technique to automatic extraction of such templates from traces of sample applications.

This dissertation defined an implementation template as a Java pseudocode that summarizes the necessary implementation steps to instantiate a given framework-provided concept. More specifically, an implementation template specifies which framework packages to import, interfaces to implement, classes to subclass, methods to implement, objects to create, methods to call, as well as some additional information such as call nesting, order of calls, and object passing patterns. This dissertation also introduced the FUDA framework comprehension technique as an automated approach for extracting such implementation templates from traces obtained by invoking concepts of interest in sample applications. FUDA was prototyped and tested on twelve concepts of four widely-used frameworks. The concept sample included both simple and complex ones. Six concepts corresponded to questions found at developer forums. The experimental evaluation illustrated that, for the considered concepts, FUDA can extract templates with relatively high precision (59-100%) and recall (79-100%) from only two traces and two sample applications per concept. Finally, in a user experiment with twelve subjects no statistically significant difference between using templates vs. documentation in terms of implementation time and number of introduced bugs could be detected. In particular, for the studied sample, the choice of templates vs. documentation had much less impact on development time than the complexity of the concept. Thus, based on this experiment, it was concluded that FUDA templates could serve as a substitute for framework documentation when no documentation is available. Nonetheless, this user study also suggested that templates should be used together with the sample applications from which they were extracted in order to obtain more information about the implementation of the concept and to detect false positives and negatives in the templates.

7.3 Future Work

This section concludes this dissertation by proposing future research directions in three areas:

- *Improving the process of FUDA and the template generation algorithms.* This dissertation showed that it is possible to automatically extract useful implementation templates for a concept of interest from example applications. Some interesting open problems are: improving the syntax of templates, rendering templates in different formats (*e.g.*, extracting templates in the form of feature models), or using different template extraction techniques (*e.g.*, applying static analysis instead of dynamic analysis).

  The current process of FUDA can be also improved. Presently, FUDA considers only the concept invoking scenarios. Nevertheless, there might be cases where using a combination of invoking and non-invoking scenarios could be more beneficial. Just for illustration purposes, consider the concept Title-bar Color presented in Table 5.1. To comprehend how to implement this concept,

one possibility would be to have two scenarios: one that changes the title-bar color and one that does not. Next, the difference between these two scenarios could be the instructions that change the color of a GEF editor's title-bar.

Another direction for future work is to improve the template generation algorithms discussed in Section 4.4.4. Some possible improvements are related to addressing the issues outlined in Section 4.4.5. These issues were problems with creating classes, their methods and constructors, and the cyclic dependencies among the program statements in the body of a method. The other interesting problem for future work which is also related to the problem of cyclic dependencies is to extract iterative patterns and framework interaction protocols from the calls. As explained in Section 4.2, FUDA templates currently do not reflect a sequence of statements that should be repeated as a block, rather than individually. By extracting such patterns and protocols, FUDA templates would contain more useful information about the implementation of a desired framework-provided concept.

- *Developing a FUDA-specific GUI.* Section 6.2.3 described that the current prototype implementation of FUDA does not provide a FUDA-specific GUI, particularly for supporting the traceability links among the templates and sample applications. Development of such a GUI could make FUDA more attractive in practise and improve the productivity of developers as explained in Section 6.2.3.

- *Performing more evaluations with more concepts and subjects.* This dissertation presented the results of an empirical study for twelve realistic concepts on top of four widely-used frameworks. Nonetheless, this evaluation could be improved by performing more experiments with non-GUI concepts, multi-threaded concepts, or some concepts on top of domain-specific frameworks that provide concepts such as data/control algorithms, transactions, and concurrency; and the frameworks that support reflection such as *Struts* and *Spring* which are quite common nowadays. To conduct these extra evaluations, it might have been required to improve the FUDA process and the template generation algorithms as well. For instance, Antoniol and Guehéneuc in [9] provide some guidelines and techniques for tracing and marking multi-threaded applications.

  Template usage evaluations could be also improved by performing experiments with more subjects to compare templates vs. framework documentation and provide more evidence whether they are actually similar in aiding the application developers or not. Finally, another interesting user experiment would be to ask a group of developers to use only sample applications as a guide to implement framework-provided concepts, and compare their results with those groups of subjects who used the documentation or the templates together with sample applications. This experiment would make the conclusions made in this dissertation stronger.

APPENDICES

Appendix A

Materials for Template Usage Evaluation

This appendix contains the supporting materials for the template usage evaluation described in Chapter 6. Section A.1 features the materials used during recruiting subjects. Section A.2 presents the tutorials used in training the subjects. Finally, Section A.3 includes the task package materials. These materials as well as the data from the study are available at [41].

A.1 Materials for Recruiting Subjects

This section features the materials used during recruiting subjects, i.e., the initial package sent to the subjects as described in Section 6.1.11. This includes the overview document and the background questionnaire.

An Overview of the FUDA Empirical Evaluation

FUDA (*Framework API Understanding through Dynamic Analysis*) is a semi-automated framework comprehension technique developed by the members of the Generative Software Development Lab[1] at the University of Waterloo, Canada with the aim of helping application developers during the process of implementing applications on top of the object-oriented software frameworks. For this purpose, FUDA generates some pseudo-Java templates that specify the implementation steps that are necessary to realize a desired concept such as a *Context Menu* on top of a desired framework such as the *Eclipse JFace*. These templates are generated from dynamic traces that are collected at runtime by invoking the concept of interest in at least two sample applications.

The intent of performing this empirical evaluation is to see to what extent the FUDA templates are comparable with the framework's documentation. For this purpose, we would like to ask you to implement two different concepts on top of a framework (either Eclipse or JFace): one using the documentation of the concept, and one using the FUDA templates. Then, we would like you to tell us about your experience.

The following steps will be followed during this empirical evaluation:

1) We will give you a questionnaire to understand about your background such as your programming experience, your familiarity with the object-oriented software frameworks, your familiarity with the Eclipse environment, and so on.
2) Based on your answers in the previous step, we will provide you some tutorials so that we can be sure you have enough knowledge about the technologies involved. These tutorials are mainly about the Eclipse environment, object-oriented software frameworks, and FUDA templates.
3) Again, based on your answers to the questionnaire in Step 1, we will assign you two concepts to implement: one with the help of the documentation of the concept, and one with the help of the FUDA templates.
    a) For each concept, the following things are provided to you in this step:
        i. A document containing a set of instructions that we would like you to follow during the implementation session.
        ii. A description of the concept that we would like you to implement.
        iii. An Eclipse project which is incomplete and we would like you to implement the assigned concept in it.
        iv. Two sample applications that provide example implementations of the desired concept.
        v. Based on how you are supposed to implement the concept, either the documentation of the concept or the FUDA template.
    b) We will define a recommended amount of time for you to try to

[1] http://gsd.uwaterloo.ca

implement the assigned concept in the provided project. Nevertheless, you may continue the implementation until either you finish it successfully or you come to the conclusion that it is impossible for you to implement the concept successfully. ***You must mark exactly how much time it took you to implement it***. This is very important for the success of the experiment.

c) All the experiments should be done in the default installation of Eclipse 3.2.2 and it is not required to install anything extra.

4) After implementing each concept, your task would be to answer a questionnaire to give us hints on your impressions about the experiment and send us your filled questionnaire together with your implementation project to aheydarnoori@uwaterloo.ca.

The results of this experiment will be published in academic publications. However, your name is completely confidential and it will not be released wherever the results of the experiment are published.

Background Questionnaire

NOTE: This is a fillable PDF file. If you are not using a PDF writer, please make sure to print this file to a PS or PDF file to not lose the information.

Name: _____

Date: _____ # Terms in Program: _____

Degree Program: ☐ Full-Time Student ☐ Part-Time Student ☐ Not a Student

If a part-time student or not a student, please specify the occupation: _____

☐ Bachelor's ☐ Master's ☐ PhD

Q.1: How many years of industrial experience do you have? _____

Q.2: In the range of 1-5, specify your familiarity with the ideas of object-orientation such as objects, classes, inheritance, etc.:

☐ 1 = None ☐ 2 ☐ 3 ☐ 4 ☐ 5 = Expert

Q.3: In the range of 1-5, specify your familiarity with the ideas of object-oriented software frameworks such as framework API, framework-provided concepts, etc.?

☐ 1 = None ☐ 2 ☐ 3 ☐ 4 ☐ 5 = Expert

Q.4: How many years of programming experience do you have in general? _____

Q.5: What programming languages are you the most expert in?

1) _____ 2) _____ 3) _____ 4) _____

Q.6: In particular, how many years of programming experience do you have in Java? _____

Q.7: In the range of 1-5, specify your skill level with the Java programming language:

☐ 1 = None ☐ 2 ☐ 3 ☐ 4 ☐ 5 = Highly Skilled

Q.8: In the range of 1-5, specify your skill level with the Eclipse environment and its Java development tools (JDT):

☐ 1 = None ☐ 2 ☐ 3 ☐ 4 ☐ 5 = Highly Skilled

Q.9: In the range of 1-5, specify your familiarity with the Eclipse framework:

☐ 1 = None ☐ 2 ☐ 3 ☐ 4 ☐ 5 = Highly Skilled

Q.9.1: Have you ever implemented the following concepts using the Eclipse framework?

Concept: Navigate[1] ☐ No ☐ Yes, Implemented before.

Concept: Table Viewer[2] ☐ No ☐ Yes, Implemented before.

Q.10: In the range of 1-5, specify your familiarity with the Eclipse JFace framework:

☐ 1 = None ☐ 2 ☐ 3 ☐ 4 ☐ 5 = Highly Skilled

Q.10.1: Have you ever implemented the following concepts using the Eclipse JFace framework?

Concept: Context Menu[3] ☐ No ☐ Yes, Implemented before.

Concept: Content Assist[4] ☐ No ☐ Yes, Implemented before.

Q.11: Do you have any other experience with other object-oriented software frameworks? If yes, please specify the framework and your skill level.

1)

2)

3)

4)

Q.12: Please provide us any questions or comments you may have about this experiment:

[1] How to create the tree navigation actions (*Go Home*, *Go Back* and *Go Into*) in the Eclipse tree viewer's toolbar?
[2] How to implement an Eclipse table viewer?
[3] How to implement a context menu in an Eclipse view using the JFace framework?
[4] How to implement a content assist in an Eclipse text editor?

A.2 Tutorials

This section presents the tutorials used in training the subjects based on their backgrounds. These tutorials include a tutorial briefly describing object-oriented application framework, a tutorial about the syntax and the semantics of FUDA templates, and a tutorial describing the necessary tasks that should be done in the Eclipse IDE during a task performance. To save space, the screenshots are removed from the Eclipse IDE tutorial.

Tutorial: Object-Oriented Application Frameworks

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JApplet;
import javax.swing.JButton;

public class ScrollingSimple
extends JApplet
implements ActionListener {

    JButton button;

    public void init() {
        button = new JButton("Click Me");
        button.addActionListener(this);
        add(BorderLayout.SOUTH, button);
    }
    public void start() {
        System.out.println("Applet Starting.");
    }
    public void stop() {
        System.out.println("Applet Stoping.");
    }
    public void actionPerformed(ActionEvent event) {
        button.setText("Click Again");
    }
}
```

Figure 1: A simple applet on top of the Java Swing and Java AWT frameworks.

An *object-oriented application framework* is a collection of classes implementing the shared architecture and the common functionality of a family of applications. Each application framework provides an *Application Programming Interface* (*API*) through which an application program can either specialize the framework code or it can directly use the framework code.

Frameworks provide *domain-specific concepts*, which are generic units of functionality. Framework-based applications are constructed by writing *completion code* that instantiates these concepts. For example, a graphical user interface (GUI) framework such as *JFace* offers implementation for a set of GUI concepts, which include a text box, tree viewer, and context menu. The instantiation of such concepts requires various *implementation steps* in the completion code, such as subclassing framework-provided classes, implementing interfaces, and calling appropriate framework services.

In short, the framework classes define the architectural skeleton to which the application program must conform. Application programs can define their own classes that may extend classes from the framework and may implement interfaces from the framework. An application program may customize and interact with a framework in ways that are permitted by object-oriented programming languages such as sub-classing, implementing interfaces, overriding superclass methods, creating instances of framework classes, and calling methods of framework classes. For example, the Java source code in Figure 1 creates a simple Java applet using the Java Swing and Java AWT frameworks. This simple applet presents a button on the bottom of the applet. Whenever the user clicks on the button, the text of the button changes from "Click Me!" to "Click Again!". The bolded sections of the source code are references to framework classes, interfaces, and methods as described in the following:

1. Subclass framework class `JApplet`.
2. Implement framework interface `ActionListener`.
3. Hold a field of type framework class `JButton`.
4. Override framework method `init` from `JApplet`. Then, within this method:
   (a) Instantiate an object of type framework class `JButton`.
   (b) Call the framework method `addActionListener` on the the `JButton`.
   (c) Read the framework constant `BorderLayout.SOUTH`.
   (d) Call the framework method `add` on the `JApplet`.
5. Override the framework method `start` from `JApplet`.
6. Override the framework method `stop` from `JApplet`.
7. Implement the framework-declared method `actionPerformed` from the framework interface `ActionListener`. Then, within this method:
   (a) Call the framework method `setText` on the `JButton`.

Tutorial: The Syntax and the Semantics of FUDA Templates

1 Definition

A template is a representation of the *implementation steps* that are necessary to instantiate a given concept. A part of a template for the concept drawing a figure in a GEF editor on top of the Eclipse GEF framework is shown in Figure 1. A FUDA template has the form of a textbook-like example in Java-based pseudo-code.

2 The Templates Content

Templates specify the following implementation steps:

- Packages to import (l. 1–4 in Figure 1);
- Framework classes to subclass (l. 5, l. 22);
- Interfaces to implement (l. 15);
- Methods to implement (l. 16);
- Objects to create (l. 7, 17, 26, 29, 31); and
- Methods to call (l. 9, 11, 12, 19, 28, 30, 32).

In addition to the basic implementation steps, the template also reflects:

- Call nesting, e.g., `setModel()` is called directly or indirectly by `createEditPart()` (l. 19);
- Call order, e.g., `paletteRoot` (l. 29) is first created before calling its `setDefaultEntry()` method (l. 30);
- Parameter passing patterns, e.g., the `defaultEditDomain` object passed to the `setEditDomain()` method (l. 32) is obtained by a prior call to **`new`** `DefaultEditDomain()` (l. 31).

Note that the specified steps involve only the elements of the framework API and implementation steps that are specific to a particular sample application are not reflected in the template.

It is also worth mentioning that the classes of the template only include the methods that are actually executed when the desired concept is invoked at runtime. For example, the body of the class `AppGraphicalNodeEditPolicy` is empty and it shows that none of its methods is called when the figure drawing concept is executed.

2.1 The Meaning of Comments

- The comments `REPEATED!` (e.g., l. 9) and `MAY REPEAT!` (l. 11) indicate that the commented step appeared more than once in every or some of the traces used to generate the template, respectively.

```
 1                 import org.eclipse.gef.editpolicies.GraphicalNodeEditPolicy;
 2                 import org.eclipse.gef.EditPart;
 3                 import org.eclipse.gef.EditPartViewer;
 4                 import org.eclipse.ui.part.WorkbenchPart;

 5  /* FTL_01 */ public class AppComponentEditPolicy extends ComponentEditPolicy {
 6  /* FTL_02 */     public Command createDeleteCommand(GroupRequest) {
 7  /* FTL_03 */         AppGraphicalNodeEditPolicy appGraphicalNodeEditPolicy =
 8                           new AppGraphicalNodeEditPolicy();
 9  /* FTL_04 */         EditPart editPart = appComponentEditPolicy.getHost(); // REPEATED!
10                           /* UNKNOWN ORDER FOR THE FOLLOWING INSTRUCTIONS */
11  /* FTL_05 */         Object object1 = editPart.getModel(); // MAY REPEAT!
12  /* FTL_06 */         EditPart editPart = editPart.getParent();
13                     }
14                 }

15  /* FTL_07 */ public class AppditPartFactory implements EditPartFactory {
16  /* FTL_08 */     public EditPart createEditPart(editPart)||(appAbstractGraphicalEditPart) {
17  /* FTL_09 */         AppAbstractGraphicalEditPart appAbstractGraphicalEditPart =
18                           new AppAbstractGraphicalEditPart();
19  /* FTL_10 */         appAbstractGraphicalEditPart.setModel(object1)||(editPart); // REPEATED!
20                     }
21                 }

22  /* FTL_11 */ public class AppGraphicalNodeEditPolicy extends GraphicalNodeEditPolicy {
23                 }

24                 public class SomeClass {
25                     public void someMethod() {
26  /* FTL_12 */         TemplateTransferDragSourceListener templateTransferDragSourceListener =
27                           new TemplateTransferDragSourceListener(EditPartViewer)||(EditPartViewer,Transfer);
28  /* FTL_13 */         EditPartViewer.addDragSourceListener(templateTransferDragSourceListener);
29  /* FTL_14 */         PaletteRoot paletteRoot = new PaletteRoot();
30  /* FTL_15 */         paletteRoot.setDefaultEntry(ToolEntry);
31  /* FTL_16 */         DefaultEditDomain defaultEditDomain = new DefaultEditDomain(IEditorPart);
32  /* FTL_17 */         GraphicalEditor.setEditDomain(defaultEditDomain);
33                     }
34                 }
```

Figure 1: Part of a template generated by FUDA for the concept drawing figures in a GEF editor

- The comments in the form of `/* FTL_n */` are used to provide traceability between the templates and the sample applications' source code from which the templates are generated. In these comments, *n* represents the number of the implementation step in the template. More specifically, all the lines of the sample applications' source code are commented with `/* FTL_n */` that correspond to the n^{th} implementation step in the template. Therefore, you can search the sample applications' source code with `FTL_n` to see how that template's implementation step is actually implemented in the provided sample applications.
- Some templates have the comment `/* UNKNOWN ORDER FOR THE FOLLOWING INSTRUCTIONS */` (e.g., l. 10). This comment shows that FUDA was not able to automatically identify an exact order for the instructions that follow this comment.

3 Differences from Ordinary Java

There are two main differences between templates and ordinary Java programs:

1. *Using the notion of '||':* FUDA uses a special syntax to show that a method with a given name was called with different argument types. For example, `setModel(object1)||(editPart)` (l. 19 in Figure 1) is due to multiple calls to `setModel()` with different arguments. As another example, l. 27 illustrates that the class `TemplateTransferDragSourceListener` can be instantiated with different types of arguments.
2. *All variables are global:* What appears to be a local variable declaration in Java, such as `object1` (l. 11), actually has global meaning in the template. For that reason, `object1` can be used as a method argument in another method scope (l. 19).

4 The class `SomeClass` and the method `someMethod` in the Template

The method `someMethod` of class `SomeClass` hosts instructions that FUDA was unable to automatically identify an exact place for them. **Please note that the instructions in this method may or may not come together in the concept's implementation.** You may specify the place of each instruction yourself, probably with the help of the given sample applications.

5 Possibility of False Positives and False Negatives

A template is an approximation of the necessary implementation steps, and it can be incomplete or unsound or both. In particular, implementation steps can be missing (*false negatives*) or unrelated steps (*false positives*) can be present in some cases. Given two sample applications, FUDA will filter out any steps that are not common to both sample applications. However, based on the experiments we did, false positives are more likely to happen than false negatives.

Furthermore, some implementation detail is still missing in a template. For example, the presented template in Figure 1 only presents the implementation steps on top of the Eclipse GEF framework. However, there might be some instructions on top of other frameworks that are necessary to see the correct functionality of the drawing figure concept. Since they are on top of other frameworks (e.g., draw2d framework), they are not included in the template. As another example, whereas the calls in lines l. 9 and l. 11 are marked as candidates to be repeated, the template does not reflect the fact that they should be repeated as a block, rather than individually. Nonetheless, the user can still extract the missing details from the actual sample code.

Tutorial: Eclipse IDE

As mentioned in the overview document sent to you before, this empirical study is supposed to be performed in the Eclipse 3.2.2 environment. Furthermore, as mentioned in that document, all the developers in this study are provided with an implementation project in which the concept should be implemented and two sample applications that actually implement the concept of interest. In this tutorial, we aim to describe the basic tasks in the Eclipse environment that developers will perform during this study:

1) Importing the projects into the workspace.
2) Running the projects.
3) Searching the projects.

One can refer to the following page for more information about using the Eclipse environment:

http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.jdt.doc.user/gettingStarted/qs-BasicTutorial.htm

1. Importing the Projects into the Workspace

1) Unzip the implementation project and the two sample applications into a local folder on your computer (e.g., *'C:\FUDA'*).
2) Run the Eclipse 3.2.2 and select *'C:\FUDA'* as the workspace path.
3) Right click in the '*Package Explorer'* view and select the '*Import…'* command from the opened context menu or you can select the '*Import…'* command from the menu *'File'* in the main menu of the Eclipse.
4) Select the *'Existing Projects into Workspace'* command from the opened window and press the *'Next'* button on the bottom of the window.
5) Then, another window will be illustrated. In this window, select the choice '*Select root directory:'* and choose *'C:\FUDA'* as the root directory. Then, you will see the implementation project and the two sample applications on the bottom of this window. Make sure to select all of them and make sure to check the option *'Copy projects into workspace'* off. Then, press the *'Finish'* button. You will see that all three projects are imported into your workspace.

2. Running the Eclipse Projects

1) Execute the *'Run…'* command either from the *'Run'* menu on the main menu or from the toolbar of the Eclipse environment.

2) The ‘Run’ window will be illustrated. On the left side of this window, double click on the *‘Eclipse Application’*. A new eclipse configuration will be generated automatically. Then, press the *‘Run’* button on the bottom right corner of the window. You will see that a new instance of the Eclipse will be executed. The next time that you would like to run a new instance of the Eclipse, you do not require creating an Eclipse runtime configuration again and you can simply push the *‘Run’* button in the Eclipse toolbar.

3) When the new instance of the Eclipse workbench is executed, you can see the Eclipse view you are working on (e.g., *Sample View*) by executing the command *Window → Show View → Other… → Sample Category → Sample View* in the new instance of the Eclipse.

3. Searching the Projects

1) You may search the projects in your workspace for a desired keyword by executing the *‘Search…’* command on the *‘Search’* menu in the main menu of the Eclipse.

2) The ‘Search’ window will be displayed. One can enter the desired keyword in this window and search the projects.

A.3 Task Package Materials

This section contains the materials included in the task packages sent to subjects for each concept implementation task. Each task package included a concept instance specification, a target application, two sample applications, either the implementation template or the documentation of the given concept, a different set of instructions based on the treatment (using templates or documentation), and a different experiment questionnaire based on the treatment (using templates or documentation). The reader is referred to Section 6.1.11 for complete description of task package materials. In the following, the set of instructions, concept instance specifications, and experiment questionnaires are presented. The implementation templates, concepts documentation, target applications, and sample applications can be obtained from the FUDA web-page [41].

Instructions for Developers Using the FUDA Templates

The implementation material includes the following things:

- *Concept Instance Description:* Provides a description of the concept we are asking you to implement and the recommended implementation time.
- *Target Project:* An Eclipse project in which you are supposed to implement the concept.
- *Sample Applications:* Two real example applications that actually implement the concept.
- *Concept Template:* Provides you a template generated automatically using the FUDA technique with the help of the provided two sample applications.
- *Experiment Questionnaire:* A questionnaire we want you to fill after your implementation session.

Please follow the following steps to perform the experiment:

1) Import the *Target Project* and *Sample Applications* into your Eclipse workspace.
2) Investigate the source code of the *Target Project* to become confident that you are familiar enough with the provided target project. We tried our best to provide you as simple target projects as possible to help you focus on the implementation of the concept. You should be able to do this investigation in a few minutes.
3) ***Before starting the implementations, you are not allowed to investigate the example applications' source code and/or read the templates.***
4) Start implementing the concept in the provided target project with the help of the provided template and sample applications. ***Write down your start time.***
5) During the implementation, please write whatever you think might be important to the experiment such as what difficulties you had with the templates? What instructions were missing? etc. However, do not waste your time on writing the descriptions. Try to use simple keywords to write your comments. Write complete descriptions after the implementation session.
6) Finish implementing the concept when you have finished implementing the concept successfully or you have come to the conclusion that you are unable to implement the concept. ***Write down your finish time.***
7) Fill the provided experiment questionnaire and email it together with your target project to aheydarnoori@uwaterloo.ca.

Note 1: Please note that time is an important factor in this experiment. Therefore, please be careful to provide us the precise amount of time you spent on implementing the concept.

Note 2: The implementation session should be done in one shot.

During the implementation you are allowed to:

- Use the provided template.
- Navigate and investigate the provided sample applications' source code if you wish. For this purpose, you may use the traceability links, i.e., the template comments in the form of */*FTL_n*/*, to trace back from the template to the sample applications' source code.
- Use the Java doc if needed.
- Use the Eclipse Java editor code completion feature.

During the implementation you are NOT allowed to:

- Use the Eclipse wizards.
- Use the Eclipse help.
- Use the Internet to search for the implementations of the concept.
- Use any other source of information that may help you implement the concept except the provided template and sample applications.

Instructions for Developers Using the Documentation

The implementation material includes the following things:

- *Concept Instance Description:* Provides a description of the concept we are asking you to implement and the recommended implementation time.
- *Target Project:* An Eclipse project in which you are supposed to implement the concept.
- *Sample Applications:* Two real example applications that actually implement the concept.
- *Documentation:* Provides you the best available documentation we were able to find for the concept. The provided documentation presents enough information required to implement the concept.
- *Experiment Questionnaire:* A questionnaire we want you to fill after your implementation session.

Please follow the following steps to perform the experiment:

1) Import the *Target Project* and the *Sample Applications* into your Eclipse workspace.
2) Investigate the source code of the *Target Project* to become confident that you are familiar with the provided target project. We tried our best to provide you as simple target projects as possible to help you focus on the implementation of the concept. You should be able to do this investigation in a few minutes.
3) ***Before starting the implementations, you are not allowed to investigate the example applications' source code and/or read the documentation.***
4) Start implementing the concept in the provided target project with the help of the provided documentation and sample applications. ***Write down your start time.***
5) During the implementation, please write whatever you think might be important to the experiment such as what difficulties you had with the provided documentation? What instructions were missing? etc. However, do not waste your time on writing the descriptions. Try to use simple keywords to write your comments. Write complete descriptions after the implementation session.
6) Finish implementing the concept when you have finished implementing the concept successfully or you have come to the conclusion that you are unable to implement the concept. ***Write down your finish time.***
7) Fill the provided experiment questionnaire and email it together with your target project to aheydarnoori@uwaterloo.ca.

Note 1: Please note that time is an important factor in this experiment. Therefore, please be careful to provide us the precise amount of time you spent on implementing the concept.

Note 2: The implementation session should be done in one shot.

During the implementation you are allowed to:

- Use the provided documentation.
- Navigate and investigate the provided sample applications' source code if you wish.
- Use the Java doc if needed.
- Use the Eclipse Java editor code completion feature.

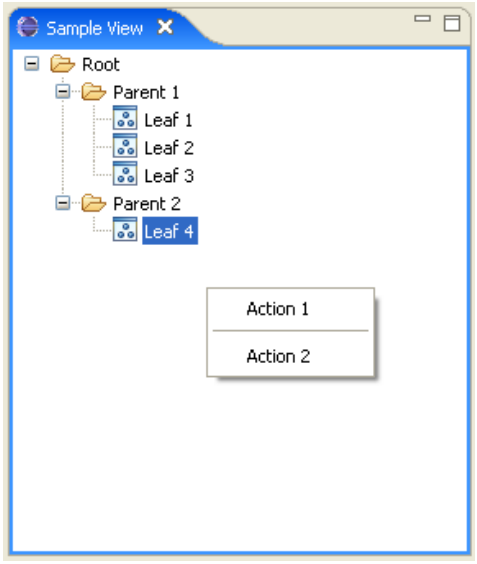
During the implementation you are NOT allowed to:

- Use the Eclipse wizards.
- Use the Eclipse help.
- Use the Internet to search for the implementations of the concept.
- Use any other source of information that may help you implement the concept except the provided documentation and sample applications.

Concept Instance Description for Concept Context Menu

The provided implementation project provides a simple Eclipse tree viewer named '*Sample View*' as shown in the following figure. You can open this view by importing this Eclipse plugin project into your Eclipse workspace and running a new Eclipse instance using the '*Run...*' command. Then, open this view using the command *Window → Show View → Other… → Sample Category → Sample View.*

In this experiment we would like you to develop the functionality of the context menu in the provided implementation project. We want this context menu to include two menu items that are labeled *Action 1* and *Action 2* and are separated by a separator. The functionality of these menu items is not important for us. Therefore, these menu items can do nothing or they can just represent a simple message on your discretion. The following figure also illustrates a snapshot of the context menu we are asking you to implement.

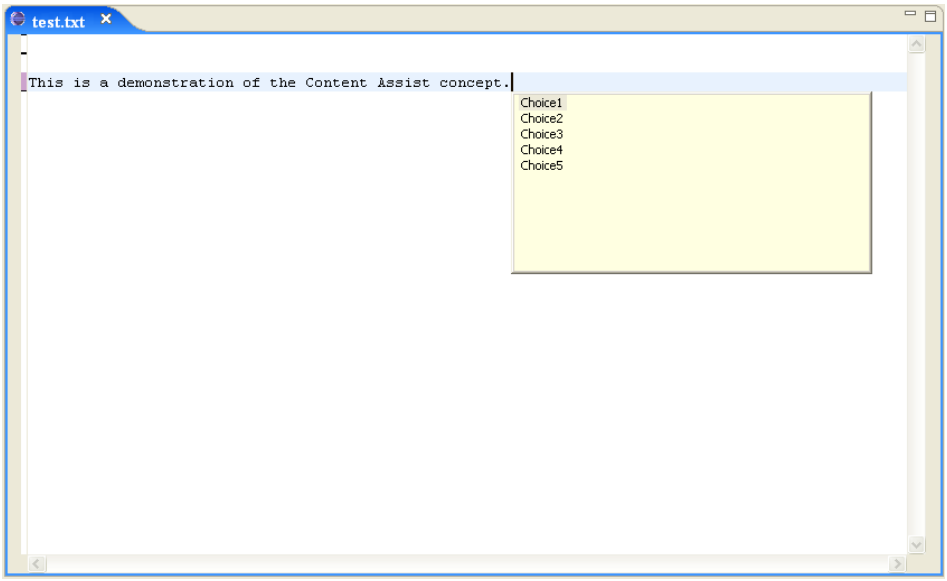


Recommended Implementation Time: 60 minutes

Concept Instance Description for Concept Content Assist

The provided implementation project implements a basic text editor named '*FUDA Text Editor*' and can be used to edit '*.txt*' files. After importing this implementation project into your Eclipse workspace and running a new Eclipse instance using the '*Run...*' command, you may r-click on a '*.txt*' file and execute the '*Open With*' → '*FUDA Text Editor*' command from the popup menu to open and edit that file.

Content assist allows users to provide context sensitive content completion upon user request. In this experiment we would like you to implement the functionality of the content assistant in the provided text editor. We want you to change the provided editor in such a way that whenever the user enters the character *dot* ('.') in the editor, a list of choices *Choice 1* – *Choice 5* to be opened from which he can select one. The following figure illustrates a snapshot of what we are asking you to implement.

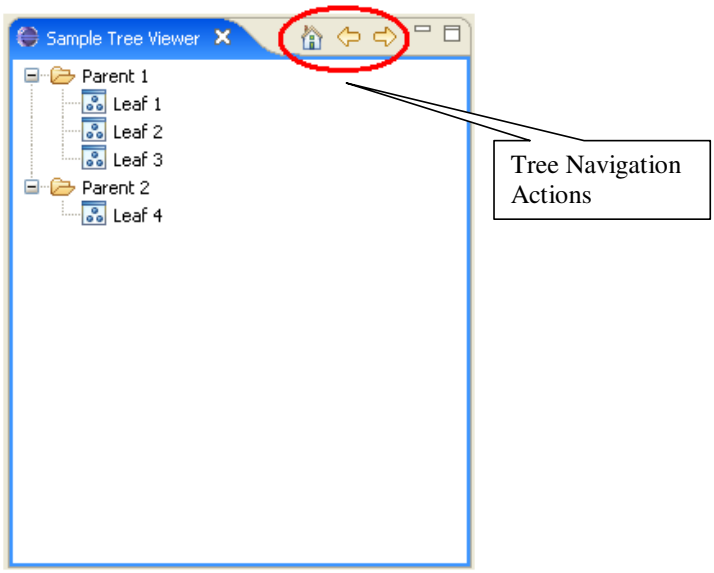


Recommended Implementation Time: 90 minutes

Concept Instance Description for Concept Navigate

The provided implementation project provides a simple Eclipse tree viewer named '*Sample Tree Viewer*' as shown in the following figure. You can open this view by importing this Eclipse plugin project into your Eclipse workspace and running a new Eclipse instance using the '*Run...*' command. Then, open this view using the command *Window → Show View → Other… → Sample Category → Sample Tree Viewer.*

The Eclipse framework provides a specific class by which developers can implement a simple web style navigation metaphor for a tree viewer. This metaphor supports the *Go Home*, *Go Back* and *Go Into* functions as shown in the following figure. In this experiment, we want you to create a toolbar for the provided Eclipse tree viewer and add these functions to it. The following figure illustrates a snapshot of what we are asking you to implement.



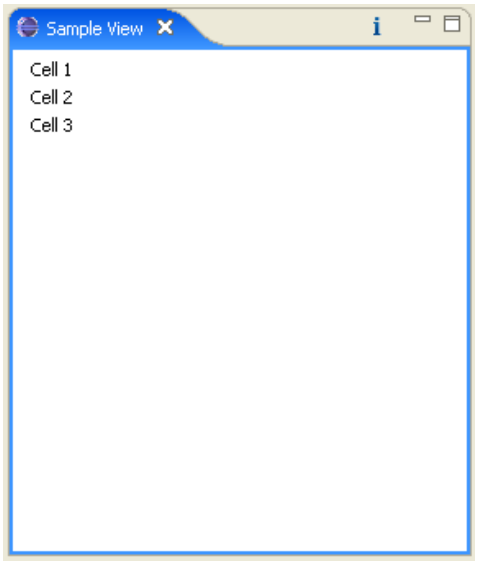
Recommended Implementation Time: 45 minutes

Concept Instance Description for Concept Table Viewer

The Eclipse framework provides constructs to develop table viewers. The provided implementation project is an incomplete Eclipse plugin project whose *'plugin.xml'* file is configured to represent a view named *'Sample Table Viewer'*.

In this experiment, we would like you to complete this project by implementing the *SampleTableViewer* class. We would like this table viewer to consist three cells labeled *Cell 1 – Cell 3*. We also would like this view to have a toolbar with only one item. The functionality of this toolbar item is not important and it can simply do nothing or it can represent a simple message on your discretion. The following figure illustrates a snapshot of what we are asking you to implement.

If you implement this concept successfully, you can open this view by running a new Eclipse instance using the '*Run...'* command. Then, open this view using the command *Window → Show View → Other… → Sample Category → Sample Table Viewer.*



Recommended Implementation Time: 90 minutes

Experiment Questionnaire for Developers Used FUDA Templates

NOTE: This is a fillable PDF file. If you are not using a PDF writer, please make sure to print this file to a PS or PDF file to not lose the information.

Name: _____ Date: _____

Concept: ☐ JFace – Context Menu ☐ JFace – Content Assist

☐ Eclipse – Navigate ☐ Eclipse – Table Viewer

Q.1: Were you able to implement the concept successfully? ☐ Yes ☐ No

Q.2: How much time did you spend on the concept's implementation? _____

Q.3: If not successful to implement the concept, what was the main reason in your opinion?

☐ Lack of experience.
☐ Not a useful template.
☐ Not useful sample applications.
☐ Complexity of the concept.
☐ Other. Please specify: _____

Q.4: Did you refer to the example applications' source code to implement the concept?

☐ No. None of them. ☐ Yes. One of them. ☐ Yes. Both of them.
Please specify: _____

Q.4.1: If yes, for what program statements and what kind of information?

Q.5: Overall, did you find the templates useful? If yes, in what way? If not, why?

Q.6: Do you think that the format and structure of the templates are OK? If not, what are the main issues?

Q.7: What kinds of information do you think are missing in the templates?

Q.8: Overall, in the range of 1-5, how do you rank the provided template in terms of usefulness to implement the concept?

☐ 1 = Not Useful ☐ 2 ☐ 3 ☐ 4 ☐ 5 = Excellent

Q.9: Do you have any additional comments on this experiment?

Experiment Questionnaire for Developers Used Documentation

NOTE: This is a fillable PDF file. If you are not using a PDF writer, please make sure to print this file to a PS or PDF file to not lose the information.

Name: _____ Date: _____

Concept: ☐ JFace – Context Menu ☐ JFace – Content Assist

☐ Eclipse – Navigate ☐ Eclipse – Table Viewer

Q.1: Were you able to implement the concept successfully? ☐ Yes ☐ No

Q.2: How much time did you spend on the concept's implementation? _____

Q.3: If not successful to implement the concept, what was the main reason in your opinion?

☐ Lack of experience.
☐ Not useful documentation.
☐ Not useful sample applications.
☐ Complexity of the concept.
☐ Other. Please specify: _____

Q.4: Did you refer to the example applications' source code to implement the concept?

☐ No. None of them. ☐ Yes. One of them. ☐ Yes. Both of them.
Please specify: _____

Q.4.1: If yes, for what program statements and what kind of information?

Q.5: How many documentations did you use in this experiment?

☐ None. ☐ Only one. ☐ Two.
Please specify: _____

Q.6: For each documentation, please specify which parts of the documentation did you read?

Doc1: ☐ None ☐ Only Relevant Parts ☐ A Big Portion ☐ All of that

Doc2: ☐ None ☐ Only Relevant Parts ☐ A Big Portion ☐ All of that

Q.7: Were you able to find all the required information for implementing the concept in the provided documentation? ☐ Yes ☐ No

Q.7.1: If not, what kinds of information were missing in the provided documentation?

Q.8: Were you able to easily access the desired information in the provided documentation? ☐ Yes ☐ No

Q.8.1: If not, what were the difficulties?

Q.9: In your opinion, was the documentation concise enough? ☐ Yes ☐ No

Q.10: Overall, in the range of 1-5, how do you rank the provided documentation in terms of usefulness to implement the concept?

☐ 1 = Not Useful ☐ 2 ☐ 3 ☐ 4 ☐ 5 = Excellent

Q.11: Do you have any additional comments on this experiment?

Appendix B

Template Generation Algorithms

This appendix presents the pseudocode of template generation algorithms discussed in Section 4.4.4.

Algorithm 1: Accepts the generalized traces and the set of common facts as input and generates a concept implementation template as output.

Input EF: Common event occurrence facts;
NF: Common event nesting facts;
DF: Common event dependency facts;
$Traces$: Set of generalized traces.
Output T: A concept implementation template.

Function createTemplate(*EF*, *NF*, *DF*, *Traces*): *T*
begin
    $T.classes \leftarrow$ createClasses(EF, DF);
    for each (class c in $T.classes$) **do**
        $c.methods \leftarrow$ createMethods(c, NF);
    end for
    createStatements($T.classes$, EF, NF, $Traces$);
    identifySupertypes($T.classes$);
    identifyVariables($T.classes$);
    broadcastVariables($T.classes$, DF);
    identifyImports(T);
    return T;
end

Algorithm 2: Creates constituent classes of the template using the sets of common event occurrence facts and common dependency facts.

Input EF: Common event occurrence facts;
DF: Common event dependency facts.
Output $\mathcal{C}$: Set of constituent classes of the template.

Function createClasses(*EF*, *DF*): $\mathcal{C}$
begin
    $\mathcal{C} \leftarrow \phi$;
    for each (group gm of incoming method calls in EF that have TT dependency in DF) **do**
        create a fresh class c;
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$;
        $c.methodCalls \leftarrow gm$;
        $c.constructorCalls \leftarrow \phi$;
    end for
    for each (outgoing constructor call $cnstrCall$ in EF) **do**
        for each (class c in $\mathcal{C}$) **do**
            if ($cnstrCall$ has RT dependency to any of the calls in $c.methodCalls$) **then**
                $c.constructorCalls \leftarrow c.constructorCalls \cup \{cnstrCall\}$;
        end for
    end for
    for each (group gc of remaining outgoing constructor calls in EF to interfaces and abstract classes that have RR dependency in DF) **do**
        create a fresh class c;
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$;
        $c.methodCalls \leftarrow \phi$;
        $c.constructorCalls \leftarrow gc$;
    end for
    return $\mathcal{C}$;
end

Algorithm 3: Creates the constituent methods and the constructor for the given class c.

Input c: A Given template class;
NF: Common event nesting facts.
Output $\mathcal{M}$: Set of methods and the constructor for class c.

1: **Function createMethods(c, NF): $\mathcal{M}$**
2: **begin**
3: $\mathcal{M} \leftarrow \phi$;
4: **for each** (method call $mtdCall$ in $c.methodCalls$) **do**
5: create a method m in class c;
6: $\mathcal{M} \leftarrow \mathcal{M} \cup \{mtdCall\}$;
7: **end for**
8: **if** ($\exists nf \in NF : nf.source \in c.constructorCalls, nf.target.direction =$ "$outgoing$") **then**
9: Create a constructor $cnstr$ for class c;
10: $\mathcal{M} \leftarrow \mathcal{M} \cup \{cnstr\}$;
11: **end**
12: **return** $\mathcal{M}$;
13: **end**

Algorithm 4: Identifies the set of statements that should go into the body of each method.

Input $\mathcal{C}$: Set of constituent classes of the template;
EF: Common event occurrence facts;
NF: Common event nesting facts;
$Traces$: Set of generalized traces.

1: **Procedure createStatements($\mathcal{C}$, EF, NF, $Traces$)**
2: **begin**
3: // Specify statements for each method.
4: **for each** (class c in $\mathcal{C} \bigwedge$ method m in c) **do**
5: $m.statements \leftarrow \{nf.target | nf \in NF, nf.source = m, nf.target.direction =$ "$outgoing$"$\}$;
6: **end for**
7: // Create the someMethod method.
8: **if** ($\exists nf \in NF : nf.source = c, nf.target.direction =$ "$outgoing$", $nf.target$ is not in the calling context of any of the methods of c) **then**
9: Create a specific method m named `someMethod` in c;
10: $m.statements \leftarrow \{nf.target | nf \in NF, nf.source = c, nf.target.direction =$ "$outgoing$", $nf.target$ is not in the calling context of any of the methods of $c\}$;
11: **end**
12: // Create the SomeClass class.
13: **if** ($\exists ef \in EF : ef.direction =$ "$outgoing$" $\bigwedge \nexists nf \in NF : nf.target = ef$) **then**
14: Create a specific class c named `SomeClass` in $\mathcal{C}$;
15: Create a method m named `someMethod` in c;
16: $m.statements \leftarrow \{ef | ef \in EF : ef.direction =$ "$outgoing$" $\bigwedge \nexists nf \in NF : nf.target = ef\}$;
17: **end**
18: // Comment the statements.
19: **for each** (class c in $\mathcal{C} \bigwedge$ method m in $c \bigwedge$ statement s in m) **do**
20: **if** (s is repeated in the calling context of m in all the input traces $Traces$) **then**
21: $s.comment \leftarrow$ "$REPEAT$";
22: **else if** (s is repeated in the calling context of m in some of the input traces $Traces$) **then**
23: $s.comment \leftarrow$ "$MAYREPEAT$";
24: **end for**
25: **end**

Algorithm 5: This algorithm identifies the supertypes, i.e., the superclass and the interfaces, of the constituent classes of the template.

Input $\mathcal{C}$: Set of constituent classes of the template.

```
1:  Procedure identifySupertypes(C)
2:  begin
3:      for each (class c except SomeClass in C) do
4:          hierarchy ← The type hierarchy of target types of method and constructor calls assigned to c,
            i.e., c.methodCalls and c.constructorCalls;
5:          c.interfaces ← φ;
6:          c.superclass ← nil;
7:          for each (leaf ℓ of the hierarchy) do
8:              if (ℓ is interface) then
9:                  c.interfaces ← c.interfaces ∪ {ℓ};
10:             else
11:                 c.superclass ← ℓ;
12:         end for
13:     end for
14: end
```

Algorithm 6: This algorithm identifies class and variable names.

Input $\mathcal{C}$: Set of constituent classes of the template.

```
1:  Procedure identifyVariables(C)
2:  begin
3:      for each (class c in C) do
4:          // Specify the class names.
5:          if (c is not SomeClass) then
6:              if (c.superclass ≠ nil) then
7:                  c.name ← "App" + c.superclass.name;
8:              else
9:                  c.name ← "App" + c.interfaces.get(0).name;
10:         end
11:         for each (method m in c) do
12:             // Specify the methods declarations.
13:             if (m is not someMethod) then
14:                 if (m has one declared signature in framework API) then
15:                     Set m's return type and parameters as what is declared in the framework API;
16:                 else
17:                     Set m's alternative return types and parameters using the '||' notation;
18:             end
19:             // Specify the statements declarations.
20:             for each (statement s in m) do
21:                 if (s has one declared signature in the framework API) then
22:                     Set s' return type and parameters as what is declared in the framework API;
23:                     Set s' return variable name as the return type, but the first letter in lower case;
24:                 end
25:                 else
26:                     Set s' alternative return types and parameters using the '||' notation;
27:                     Set s' return variable name as one of the return types, but the first letter in lower
                        case;
28:                 end
29:             end for
30:         end for
31:     end for
32: end
```

Algorithm 7: This algorithm broadcasts variable declarations among the program statements based on the dependency types.

Input $\mathcal{C}$: Set of constituent classes of the template;
DF: Common event dependency facts.

Procedure broadcastVariables($\mathcal{C}$, DF)
begin
    $G \leftarrow$ A graph in which nodes are the statements in $\mathcal{C}$, edges are dependency facts in DF except TR, PR, and RR facts;
    if (G is not cyclic) **then**
        topologicalSort(G);
        Broadcast variables along the edges of G according to dependency facts;
    end
    else
        for each (class c in $\mathcal{C} \bigwedge$ method m in c) **do**
            $g \leftarrow$ Subgraph of G that corresponds to m;
            if (g is not cyclic) **then**
                topologicalSort(g);
                Broadcast variables along the edges of g according to dependency facts;
            end
        end for
    end
end

Algorithm 8: This algorithm creates the list of package imports.

Input T: Template.

Procedure identifyImports(T)
begin
    $T.imports \leftarrow \phi$;
    Remove package names from the fully qualified names of non-primitive types and add them to $T.imports$;
end

References

[1] Mithun Acharya, Tao Xie, Jian Pei, , and Jun Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2007. 20, 21

[2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the 1990 Conference on Programming Language Design and Implementation (PLDI)*, pages 246–256, 1990. 4, 30

[3] Matthew Allen and Susan Horwitz. Slicing Java programs that throw and catch exceptions. In *Proceedings of the 2003 Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 44–54, 2003. 30

[4] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages (POPL)*, pages 98–109, 2005. 21

[5] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL)*, pages 4–16, 2002. 20, 21

[6] Michal Antkiewicz, Thiago Tonelli Bartolomei, and Krzysztof Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *Proceedings of the 22nd Conference on Automated Software Engineering (ASE)*, 2007. 17

[7] Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 692–706, 2006. 17

[8] Giuliano Antoniol and Yann-Gael Guehénéuc. Feature identification: A novel approach and a case study. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pages 357–366, 2005. 26, 27

[9] Giuliano Antoniol and Yann-Gael Guehénéuc. Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, 32(9):627–641, 2006. 23, 26, 27, 123

[10] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 206–222, 1993. 30

[11] Silvia Breu. Extending dynamic aspect mining with static information. In *Proceedings of the 5th Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 57–65, 2005. 28

[12] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *Proceedings of the 19th Conference on Automated Software Engineering (ASE)*, pages 310–315, 2004. 28

[13] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *Proceedings of the 21st Conference on Automated Software Engineering (ASE)*, pages 221–230, 2006. 28

[14] Marcel Bruch, Thorsten Schäfer, and Mira Mezini. FrUiT: IDE support for framework understanding. In *Proceedings of the 4th OOPSLA Workshop on Eclipse Technology Exchange (eTX)*, pages 55–59, 2006. 2, 18

[15] Magiel Bruntink, Arie van Deursen, Tom Tourwe, and Remco van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM)*, pages 200–209, 2004. 28

[16] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11/12):595–607, 1998. 29

[17] Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC)*. 23

[18] Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, 1994. 30

[19] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 5–14, 2007. 20

[20] Wim Codenie, Koen De Hondt, Patrick Steyaert, and Arlette Vercammen. From custom applications to domain-specific frameworks. *Communications of the ACM*, 40(10):70–77, 1997. 15

[21] Wesley Coelho and Gail C. Murphy. Presenting crosscutting structure with active models. In *Proceedings of the 5th Conference on Aspect-Oriented Software Development (AOSD)*, pages 158–168, 2006. 23, 24

[22] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. 17

[23] Barthélémy Dagenais and Harold Ossher. Aiding evolution with concern-oriented guides. In *Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution (LATE)*, page 4, 2007. 20

[24] Barthélémy Dagenais and Harold Ossher. Automatically locating framework extension examples. In *Proceedings of the 16th International Symposium on the Foundations of Software Engineering (FSE)*, pages 203–213, 2008. 20

[25] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *Proceedings of the 4th ICSE Workshop on Dynamic Analysis (WODA)*, pages 17–24, 2006. 22

[26] Danny Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pages 389–398, 2005. 69

[27] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In Forrest Shull, Janice Singer, and Dag I.K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*. Springer, 2007. 82

[28] Michael Eichberg, Michael Haupt, Mira Mezini, and Thorsten Schäfer. Comprehensive software understanding with SEXTANT. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pages 315–324, 2005. 23, 24

[29] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003. 23, 26

[30] Andrew D. Eisenberg and Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pages 337–346, 2005. 23, 26

[31] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001. 20

[32] George Fairbanks, David Garlan, and William Scherlis. Design fragments make using frameworks easier. In *Proceedings of the 2006 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 75–88, 2006. 16

[33] Mohamed E. Fayad, Ralph E. Johnson, and Douglas C. Schmidt. *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* Addison-Wesley, 1999. 8, 9, 12, 13

[34] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. UML-F: A modeling language for object-oriented frameworks. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*, pages 63–82, 2000. 16

[35] Gary Froehlich, H. James Hoover, Ling Liu, and Paul Sorenson. Hooking into object-oriented application frameworks. In *Proceedings of the 19th International Conference on Software Engineering (ICSE)*, pages 491–501, 1997. 16

[36] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th International Symposium on the Foundations of Software Engineering (FSE)*, pages 339–349, 2008. 22

[37] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 51–60, 2008. 22

[38] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins.* Addison-Wesley, 2003. 1, 18

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. 1, 7, 13, 16

[40] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer, 1999. 19, 26

[41] Generative Software Development Lab. FUDA supporting material. `http://gsd.uwaterloo.ca/~aheydarn/fuda/`, 2008. 65, 82, 117, 125, 137

[42] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification.* Addison-Wesley, 3rd edition, 2005. 42

[43] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 265–274, 2001. 25

[44] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, pages 2–27, 2006. 23

[45] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, and Jukka Viljamaa. Annotating reusable software architectures with specialization patterns. In *Proceedings of the 2nd Working Conference on Software Architecture (WICSA)*, page 171, 2001. 16

[46] R. J. Hall. Automatic extraction of executable program subsets by simultaneous program slicing. *Journal of Automated Software Engineering*, 2(1):33–53, 1995. 29

[47] Jan Hannemann and Gregor Kiczales. Overcoming the prevalent decomposition of legacy code. In *Proceedings of ICSE Workshop on Advanced Separation of Concerns in Software Engineering (ASOC)*, 2001. 28

[48] Juha Hautamäki. *Pattern-Based Tool Support for Frameworks Towards Architecture-Oriented Software Development Environment*. PhD thesis, Tampere University of Technology, Finland, 2005. 12

[49] Larry V. Hedges. Distribution theory for Glass's estimator of effect size and related estimators. *Journal of Educational Statistics*, 6(2):107–128, 1981. 100

[50] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *Proceedings of the 5th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 31–40, 2005. 20, 21

[51] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In *Proceedings of the 12th International SPIN Workshop on Model Checking Software (SPIN)*, pages 235–239, 2003. 21

[52] Abbas Heydarnoori and Krzysztof Czarnecki. Comprehending implementation recipes of framework-provided concepts through dynamic analysis. In *OOPSLA Companion, Poster Session*, 2007. 64

[53] Abbas Heydarnoori and Krzysztof Czarnecki. Comprehending object-oriented software frameworks through dynamic analysis. Technical Report CS-2007-18, University of Waterloo, 2007. Available at: `http://www.cs.uwaterloo.ca/research/tr/2007/`. 64

[54] Abbas Heydarnoori and Krzysztof Czarnecki. Mining implementation recipes of framework-provided concepts in dynamic framework API interaction traces. In *OOPSLA Companion, Tool Demonstration Track*, 2007. 64

[55] Abbas Heydarnoori, Krzysztof Czarnecki, and Thiago Tonelli Bartolomei. Supporting framework use via automatically extracted concept-implementation templates. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, 2009. 3, 33

[56] Raphael Hoffmann, James Fogarty, and Daniel S. Weld. Assieme: Finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th Symposium on User Interface Software and Technology (UIST)*, 2007. 20

[57] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 117–125, 2005. 2, 18

[58] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990. 30

[59] Daqing Hou, H. James Hoover, and Piotr Rudnicki. Specifying framework constraints with FCL. In *CASCON*, pages 96–110, 2004. 17

[60] Daqing Hou, H. James Hoover, and Changyu Yin. The framework use problem: A preliminary study with GUI frameworks. In *Proceedings of the Midwest Software Engineering Conference (midWestSE)*, 2003. 1

[61] Daqing Hou, Kenny Wong, and H. James Hoover. What can programmer questions tell us about frameworks? In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC)*, pages 87–96, 2005. 1

[62] Robert A. Jacobs. Methods for combining experts' probability assessments. *Neural Computation*, 7(5):867–888, 1995. 27

[63] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd Conference on Aspect-Oriented Software Development (AOSD)*, pages 178–187, 2003. 23, 24

[64] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings of the 1992 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 63–76, 1992. 15, 16

[65] Ralph E. Johnson. Patterns and frameworks. pages 375–382, 1998. 16

[66] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002. 28

[67] M. Kamkar. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD thesis, Linkoping University, 1993. 30

[68] M. Kamkar, P. Fritzson, and N. Shahmerhi. Interpocedural dynamic slicing applied to interprocedural data flow testing. In *Proceedings of the 9th International Conference on Software Maintenance (ICSM)*, pages 386–395, 1993. 31

[69] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997. 28

[70] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002. 80, 81

[71] B. Korel. Identifying faulty modifications in software maintenance. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 341–356, 1993. 31

[72] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988. 30

[73] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990. 29

[74] B. Korel and J. Rilling. Dynamic program slicing in understanding of program execution. In *Proceedings of the 5th International Workshop on Program Comprehension (IWPC)*, pages 80–90, 1997. 31

[75] B. Korel and J. Rilling. CASE and dynamic program slicing in software maintenance. *International Journal of Computer Science and Information Management*, 1998. 31

[76] B. Korel and J. Rilling. Dynamic program slicing methods. *Information and Software Technology*, 40(11/12):647–659, 1998. 30, 31

[77] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, 1988. 15

[78] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 5th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 306–315, 2005. 20, 21

[79] Chang Liu, En Ye, and Debra J. Richardson. Software library usage pattern extraction using a software model checker. In *Proceedings of the 21st Conference on Automated Software Engineering (ASE)*, pages 301–304, 2006. 20, 21

[80] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the 22nd Conference on Automated Software Engineering (ASE)*, pages 234–243, 2007. 2, 23, 26, 27

[81] Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *Proceedings of the 5th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 296–305, 2005. 20

[82] David Lo and Siau-Cheng Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, pages 51–60, 2006. 20, 22

[83] David Lo and Siau-Cheng Khoo. SMArTIC: towards building an accurate, robust and scalable specification miner. In *Proceedings of the 14th International Symposium on the Foundations of Software Engineering (FSE)*, pages 265–275, 2006. 20, 22

[84] David Lo, Siau-Cheng Khoo, and Chao Liu. Efficient mining of iterative patterns for software specification discovery. In *Proceedings of the 13th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 460–469, 2007. 20, 22, 58

[85] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 48–61, 2005. 18

[86] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, pages 214–223, 2004. 23, 25, 27

[87] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, pages 132–141, 2004. 28

[88] Matthias Meusel, Krzysztof Czarnecki, and Wolfgang Köpf. A model for structuring user documentation of object-oriented frameworks using patterns and hypertext. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, pages 496–510, 1997. 15

[89] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 167–176, 2000. 18

[90] Douglas C. Montgomery. *Design and Analysis of Experiments*. Wiley, 6th edition, 2004. 99, 100

[91] Alvaro Ortigosa and Marcelo Campo. SmartBooks: A step beyond active-cookbooks to aid in framework instantiation. In *Proceedings of the 29th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, page 131, 1999. 15

[92] Alvaro Ortigosa, Marcelo Campo, and Roberto Moriyón. Towards agent-oriented assistance for framework instantiation. *SIGPLAN Notices*, 35(10):253–263, 2000. 15

[93] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the 1st Software Engineering Symposium on Practical Software Development Environments (SESPSDE)*, pages 177–184, 1984. 29

[94] Jian Pei, Jian Liu, and Ke Wang. Discovering frequent closed partial orders from strings. *IEEE Transactions on Knowledge and Data Engineering*, 18(11):1467–1481, 2006. Haixun Wang and Jianyong Wang and Philip S. Yu. 21

[95] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007. 26, 27

[96] Denys Poshyvanyk, Andrian Marcus, Vaclav Rajlich, Yann-Gael Guehéneuc, and Giuliano Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC)*, pages 137–148, 2006. 27

[97] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995. 16

[98] Wolfgang Pree, Gustav Pomberger, Albert Schappert, and Peter Sommerlad. Active guidance of framework development. *Software - Concepts and Tools*, 16(3):136–145, 1995. 15

[99] Tao Qin, Lu Zhang, Zhiying Zhou, Dan Hao, and Jiasu Sun. Discovering use cases from source code using the branch-reserving call graph. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference (APSEC)*, page 60, 2003. 25

[100] Murali K. Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 240–250, 2007. 2, 21

[101] Murali K. Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In *Proceedings of the 2007 Conference on Programming Language Design and Implementation (PLDI)*, pages 123–134, 2007. 20, 21

[102] Don Roberts and Ralph Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Proceedings of the 3rd Conference On Pattern Languages of Programs (PLoP)*, 1996. 7

[103] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 406–416, 2002. 23, 24

[104] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):3–38, 2007. 23, 24

[105] Martin P. Robillard and Frédéric Weigand-Warr. ConcernMapper: simple view-based separation of scattered concerns. In *Proceedings of the 3rd OOPSLA Workshop on Eclipse Technology Exchange (eTX)*, pages 65–69, 2005. 23, 24

[106] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: Mining for sample code. In *Proceedings of the 2006 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 413–430, 2006. 18

[107] Maher Salah and Spiros Mancoridis. A hierarchy of dynamic software views: From object-interactions to feature-interactions. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM)*, pages 72–81, 2004. 26, 27

[108] Maher M. Salah. *An Environment for Comprehending the Behavior of Software Systems*. PhD thesis, Drexel University, 2005. 26, 27, 38

[109] Sriram Sankaranarayanan, Franjo Ivanči, and Aarti Gupta. Mining library specifications using inductive logic programming. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 131–140, 2008. 20, 22

[110] Thorsten Schäfer, Ivica Aracic, Matthias Merz, Mira Mezini, and Klaus Ostermann. Clustering for generating framework top-level views. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE)*, pages 239–248, 2007. 19

[111] Thorsten Schäfer, Michael Eichberg, Michael Haupt, and Mira Mezini. The SEXTANT software exploration tool. *IEEE Transactions on Software Engineering*, 32(9):753–768, 2006. 23, 24

[112] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th Conference on Aspect-Oriented Software Development (AOSD)*, pages 212–224, 2007. 23, 25

[113] David Shepherd, Jeffrey Palm, Lori Pollock, and Mark Chu-Carroll. Timna: a framework for automatically combining aspect mining analyses. In *Proceedings of the 20th Conference on Automated Software Engineering (ASE)*, pages 184–193, 2005. 28

[114] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, pages 174–184, 2007. 20

[115] Sharon Simmons, Dennis Edwards, Norman Wilde, Josh Homan, and Michael Groble. Industrial tools for the feature location problem: an exploratory study. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(6):457–474, 2006. 26

[116] Suresh Thummalapenta and Tao Xie. PARSEWeb: a programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd Conference on Automated Software Engineering (ASE)*, pages 204–213, 2007. 18

[117] Suresh Thummalapenta and Tao Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proceedings of the 23rd Conference on Automated Software Engineering (ASE)*, 2008. 19

[118] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995. 31

[119] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, pages 112–121, 2004. 28

[120] Tom Tourwé. *Automated Support for Framework-Based Software Evolution.* PhD thesis, Vrije Universiteit, 2002. 16

[121] Tom Tourwé and Kim Mens. Mining aspectual views using formal concept analysis. In *Proceedings of the 4th Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 97–106, 2004. 28

[122] Tom Tourwé and Tom Mens. Automated support for framework-based software evolution. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, page 148, 2003. 16

[123] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. LCM ver.3: Collaboration of array, bitmap and prefix tree for frequent itemset mining. In *Proceedings of the 1st Open Source Data Mining Workshop on on Frequent Pattern Mining Implementations (OSDM)*, pages 77–86, 2005. 68

[124] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the 1999 Conference on Programming Language Design and Implementation (PLDI)*, pages 107–119, 1991. 29

[125] Jukka Viljamaa. Reverse engineering framework reuse interfaces. In *Proceedings of the 4th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 217–226, 2003. 1, 16, 19

[126] Jukka Viljamaa. *Applying Formal Concept Analysis to Extract Framework Reuse Interface Specifications from Source Code*. PhD thesis, University of Helsinki, Finland, 2004. 13, 19

[127] J. Vlissides. Protection, part i: The hollywood principle. *C++ Report*, 8(2):14–19, 1996. 8

[128] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit, 1998. 24

[129] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 512–521, 2004. 31

[130] Tao Wang and Abhik Roychoudhury. Hierarchical dynamic slicing. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, pages 228–238, 2007. 31

[131] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2007. 21

[132] Ben Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–113, 1974. 20

[133] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 461–476, 2005. 22

[134] Mark Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, 1979. 29

[135] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982. 29

[136] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984. 29

[137] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*, pages 218–228, 2002. 20, 21

[138] A Taligent Inc. white paper. Building object-oriented frameworks. `http://lhcb-comp.web.cern.ch/lhcb-comp/Components/postscript/buildingoo.pdf`, 1994. 7

[139] Norman Wilde and Michael C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practise*, 7(1):49–62, 1995. 23, 26

[140] W. Eric Wong, Swapna S. Gokhale, and Joseph R. Horgan. Locating program features by using execution slices. In *Proceedings of the 2nd Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET)*, pages 194–203, 1999. 23, 26

[141] Tao Xie and Jian Pei. MAPO: Mining API usages from open source repositories. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR)*, pages 54–57, 2006. 20

[142] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 282–291, 2006. 2, 20, 22

[143] Robert K. Yin. *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. SAGE Publications, 3rd edition, 2003. 82

[144] Mohammed J. Zaki. Mining non-redundant association rules. *Data Mining Knowledge Discovery*, 9(3):223–248, 2004. 68

[145] Charles Zhang and Hans-Arno Jacobsen. Prism is research in aspect mining. In *OOPSLA Companion, Tool Demonstration Track*, 2004. 25

[146] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 167–176, 2003. 30

[147] Wei Zhao, Lu Zhang, Dan Hao, Hong Mei, and Jiasu Sun. Alternative scalable algorithms for lattice-based feature location. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM)*, page 528, 2004. 27

[148] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. SNIAFL: Towards a static non-interactive approach to feature location. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 293–303, 2004. 23

[149] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. SNIAFL: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology*, 15(2):195–226, 2006. 2, 23, 25