Two Studies of Framework-Usage Templates Extracted from Dynamic Traces

Abbas Heydarnoori, Krzysztof Czarnecki, Walter Binder, and Thiago Tonelli Bartolomei

Abstract—Object-oriented frameworks are widely used to develop new applications. They provide reusable concepts that are instantiated in application code through potentially complex implementation steps such as subclassing, implementing interfaces, and calling framework operations. Unfortunately, many modern frameworks are difficult to use because of their large and complex APIs and frequently incomplete user documentation. To cope with these problems, developers often use existing framework applications as a guide. However, locating concept implementations in those sample applications is typically challenging due to code tangling and scattering. To address this challenge, we introduce the notion of *concept-implementation templates*, which summarize the necessary concept-implementation steps and identify them in the sample application code, and a technique, named *FUDA*, to automatically extract such templates from dynamic traces of sample applications. This paper further presents the results of two experiments conducted to evaluate the quality and usefulness of FUDA templates. The experimental evaluation of FUDA with 14 concepts in five widely used frameworks suggests that the technique is effective in producing templates with relatively few false positives and false negatives for realistic concepts by using two sample applications. Moreover, we observed in a user study with 28 programmers that the use of templates reduced the concept-implementation time compared to when documentation was used.

Index Terms—Object-oriented application frameworks, framework comprehension, framework documentation, concept-implementation templates, application programming interface (API), dynamic analysis, concept location, feature identification.

1 Introduction

Object-oriented Application Frameworks have been shown to be one of the most effective reuse technologies available today, enabling reuse of both design and code [1]. Frameworks provide *domain-specific concepts*, which are generic units of functionality. Framework-based applications are constructed by writing *application code*, which instantiates these concepts. For example, the *Eclipse* framework offers concepts such as tree viewers and text editors. Eclipse's Ant View and Java Editor are instances of these concepts. The instantiation of such concepts requires following different *implementation steps* in the application code, such as subclassing framework-provided classes, implementing interfaces, and calling appropriate framework methods. These steps are governed by the framework's *application programming interface* (API).

Unfortunately, many existing frameworks are difficult to use because of their large and complex APIs and often incomplete user documentation [2]. To ease these problems, developers frequently use existing framework applications as a guide to understand how to implement a desired concept [1]. However, locating the concept implementation in existing applications can be a challenge because its implementation is often scattered across the source code and tangled with the code implementing other concepts. To cope with this challenge, feature location and identification techniques such as PROMESIR [3], SNIAFL [4], and SITIR [5] can be used to locate the parts of the application source code implementing a functionality of interest, specified by usage scenarios or domain terms. However, the difficulty with these techniques is that they do not focus on framework API usage and the code identified will still include many application-specific programming elements that are irrelevant from the viewpoint of framework usage.

Several categories of tools have been proposed in the literature that focus on API understanding. *Code assistants* such as *XSnippet* [6] and *FrUiT* [7] apply static analysis to the source code of sample applications and allow retrieving code snippets or usage rules for a particular API element in the context of a programming task at hand. They require the developer to know at least some names of the API elements needed for the concept implementation; they are less helpful if the developer has only a high-level idea of the concept to be implemented or if the concept spans multiple classes or both. *API comprehension* tools like *Pattern Extractor* [8] and *SpotWeb* [9] extract API usage patterns from sample applications to provide a general characterization of the whole framework API rather than identifying the relevant concept-implementation steps. Finally, *framework evolution comprehension* tools such as *AURA* [10] and *SemDiff* [11] analyze API changes over the framework evolution to help adapt existing applications to the changes in the new framework API release, but they do not aid implementing concepts from scratch.

To help developers implement framework-provided concepts, we introduced 1) the notion of *concept*

- *A. Heydarnoori is with the Department of Computer Engineering, Sharif University of Technology, Azadi Ave., Tehran, Iran. E-mail: heydarnoori@sharif.edu.*
- *K. Czarnecki and T.T. Bartolomei are with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L 3G1, Canada. E-mail: {kczarnec, ttonelli}@gsd.uwaterloo.ca.*
- *W. Binder is with the Faculty of Informatics, University of Lugano, Via Giuseppe Buffi 13, 6904 Lugano, Switzerland. E-mail: walter.binder@usi.ch.*

Manuscript received 21 May 2010; revised 21 June 2011; accepted 16 July 2011; published online 27 July 2011.
Recommended for acceptance by H. Gall.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-05-0154.
Digital Object Identifier no. 10.1109/TSE.2011.77.

implementation templates, which explain framework usage, and 2) *Framework API Understanding through Dynamic Analysis (FUDA)*, an approach for the automatic extraction of such templates from traces of sample applications [12], [13]. A concept implementation template is a piece of Java pseudocode summarizing the *implementation steps* for instantiating a given concept, such as which packages to import, framework classes to subclass, interfaces to implement, and operations to call. Although templates can be used on their own, our user study has shown that they are most effective when used as an entry point to explore concept implementations in sample applications.

The template extraction approach works by invoking instances of a concept of interest in two or more different contexts, located in one or more sample applications, and recording all runtime interactions between the application code and the framework API. For example, given that the context menu in an Eclipse view is the desired concept, each trace could be collected by invoking a context menu in a different Eclipse view. The collected traces then provide the basis for automatically generating the template. Because the selection of sample applications and collection of traces represent manual effort and the installation and execution of the applications can be challenging, we aim at keeping the number of traces as small as possible—which is two for FUDA.

We have implemented FUDA as a tool for Java and used it in a study to generate templates for 14 concepts from five widely used frameworks. The study shows that FUDA can produce templates with relatively few false positives (i.e., incorrect implementation steps) and false negatives (i.e., missing implementation steps) for realistic concepts by using two sample applications. We also conducted a user experiment with 28 subjects comparing templates to framework documentation as concept-implementation aids. For the studied sample, we observed that the choice of templates reduced the implementation time with statistical significance. Moreover, the analysis of additional data and feedback suggested that templates should be used together with the sample applications from which they were extracted rather than just by looking at the templates alone.

With respect to the earlier version of the work [12], the contributions of this paper include:

1. comprehensive presentation of the experiments evaluating the quality and usefulness of FUDA templates;
2. extension of the FUDA tool to support frameworks located in the standard Java class library, such as Java Swing;
3. evaluation of the quality of the generated templates for an additional framework and a set of concepts; and
4. repeating the user study with additional subjects to obtain statistically significant results.

The remainder of this paper is organized as follows: We first provide a motivating example, used throughout this paper (Section 2). We then introduce concept-implementation templates (Section 3) and describe the FUDA technique (Section 4) and its implementation (Section 5). Next, we evaluate the quality of FUDA templates (Section 6) and describe the user experiment comparing the effectiveness of templates versus framework documentation in aiding application developers (Section 7). Finally, we discuss several aspects of FUDA (Section 8), compare it with related work (Section 9); and conclude (Section 10).

2 Motivating Example

We clarify the problem that FUDA aims to tackle using an example. Consider the case of a developer creating a plug-in for the Eclipse platform. 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 in documentation, mailing lists, and newsgroups. The required steps for implementing the concept might not be documented there, however. The other option is to study the code of plug-ins that implement menus to locate their relevant parts. Unfortunately, even though the concept of a context menu has a crisp manifestation in the graphical user interface (GUI), its implementation code can be scattered and tangled with the code implementing other concepts. As a result, locating the relevant code can be challenging.

Fig. 1 represents the code implementing a context menu using the JFace framework. This code was generated using one of Eclipse's wizards. The menu is located in `SampleView`, which is a visual component that displays trees using a `TreeViewer` (l. 36). The lines implementing the context menu are marked by •. The lines marked by ◦ implement a Welcome window and 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()` (l. 190). In particular, this method calls `makeActions()` (l. 198) and `hookContextMenu()` (l. 199), 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 `menuAboutToShow()` (l. 205) callback method, which is called by the framework, i.e., JFace, when the user clicks to open the context menu.

The context menu example presents some of the difficulties in locating the implementation of framework-provided concepts in application code. As can be seen in Fig. 1, the implementation of the menu is tangled with the implementation of the view and it involves a sophisticated interaction among several objects, namely, view, menu manager, menu listener, menu, actions, and separators. To complicate the matter, a concept implementation may also be scattered across several classes, as in the case of Eclipse's drag&drop. Consequently, even though locating a concept in the graphical user interface of a sample application may be easy, locating its implementation in the application code is challenging and time consuming. Concept-implementation templates aim to address this difficulty.

3 Concept Implementation Templates

Fig. 2 shows a template for our context menu example. We extracted the template from two traces collected by

```
    ...
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());
•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 }
```

Fig. 1. Implementation of a sample Eclipse view with a context menu (•).

invoking the context menu in two sample applications: `SampleView` (Fig. 1) and `Console`, which is part of Eclipse. The template is expressed in a Java-based pseudocode, close to what the developer must write to implement the concept.

Templates specify the following implementation steps: packages to import (l. 1-8 in Fig. 2), framework classes to subclass (l. 15), interfaces to implement (l. 9), methods to implement (l. 10), objects to create (e.g., l. 11), and methods to call (e.g., l. 12). The specified steps involve only the elements of the framework API. For example, the method calls `fillContextMenu()` and `makeActions()` in Fig. 1 are specific to that particular implementation and are not reflected in the template. The code elements corresponding to these steps may be entirely *framework-defined*, e.g., the implementation of `Separator`, instantiated in line 11, resides in the framework code. Alternatively, the elements may also reside in the application

```
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 {
10     public void menuAboutToShow(menuManager) {
11         Separator separator = new Separator(String)||(); //REPEAT
12         menuManager.add(separator)||(appAction); //REPEAT
13     }
14 }

15 public class AppAction extends Action {
16 }

17 public class SomeClass {
18     public void someMethod() {
19         Viewer viewer = ...;
20         Control control = viewer.getControl(); //MAY REPEAT
21         AppAction appAction = new AppAction(); //MAY REPEAT
22         appAction.setText(String); //MAY REPEAT
23         appAction.setToolTipText(String); //MAY REPEAT
24         MenuManager menuManager = new MenuManager(String)||();
25         menuManager.setRemoveAllWhenShown(boolean);
26         AppMenuListener appMenuListener = new AppMenuListener();
27         menuManager.addMenuListener(appMenuListener);
28         Menu menu = menuManager.createContextMenu(control);
29     }
30 }
```

Fig. 2. A sample implementation template extracted for the concept context menu.

code, provided that they are *framework-stipulated*. In particular, such elements are 1) application classes that are subtypes of API-defined types, 2) application methods that implement API-defined operations or override API-defined methods, and 3) constructors of classes that are subtypes of framework classes and interfaces. 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 API-defined class `Action` in the application code.

In addition to the basic implementation steps, the template also reflects 1) call nesting, e.g., `add()` is called directly or indirectly by `menuAboutToShow()` (l. 12), 2) call order, e.g., the menu listener is added to the menu manager (l. 27) before creating the menu (l. 28), and 3) parameter passing patterns, e.g., the control object passed to the menu creation method (l. 28) 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 main exceptions. First, the code uses the notation "||" to show that a method with a given name was called with different argument types. For example, `add(separator) || (appAction)` (l. 12 in Fig. 2) is due to multiple calls to `add()` with different arguments (l. 213 and 215 in Fig. 1). Second, what appears to be a local variable declaration, 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 required and sufficient implementation steps and, thus, it can be incomplete, unsound, or both. In particular, steps can

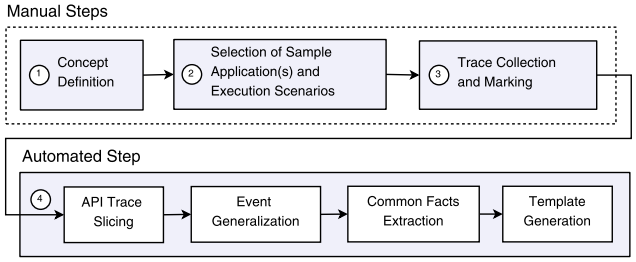


Fig. 3. FUDA process.

be missing (*false negatives*) or unrelated steps (*false positives*) can be present. Given two traces, FUDA will filter out any steps that are not common to both traces. If a necessary implementation step, say component registration, can be achieved in more than one way, e.g., by calling different methods, it will get filtered out, resulting in a false negative in the generated template. Furthermore, FUDA relies on the assumption that input traces show the execution of the concept of interest in different contexts; otherwise it may introduce false positives. Finally, some implementation details are still missing in a template. For example, although the calls in lines 21-23 are marked as candidates to be repeated, the template does not reflect 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. Traceability links between the steps in the template and the corresponding steps in the application code can support this task. We left the development of such traceability links as future work in our FUDA implementation (cf. Section 5).

4 Template Creation Using FUDA

4.1 Overview

Before delving into the details of FUDA, we first provide an overview of the technique. FUDA uses a dynamic analysis to determine the portions of application code that are relevant for implementing a concept of interest. From the user's perspective, FUDA has four steps, depicted in Fig. 3. The first three steps are manual; the fourth one consists of several automated substeps.

The first step is to determine the concept for which an implementation template should be created. The second step is to select one or more sample applications and execution scenarios that invoke the desired concept in different contexts. Two execution scenarios are typically enough, as will be shown in Section 6. The third step is to run each application under a *tracer* tool and exercise the scenarios. The tool collects traces 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 a trace, the user informs the tool of the moments right before and after the concept invocation. Finally, in the fourth step, an *analyzer* examines the collected traces and generates the implementation template. We describe each step in detail in the following sections.

4.2 Concept Definition

A prerequisite for applying FUDA is the ability to invoke the concept of interest in sample applications from their graphical or programmatic interfaces. 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 of the objects). For example, 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?*"

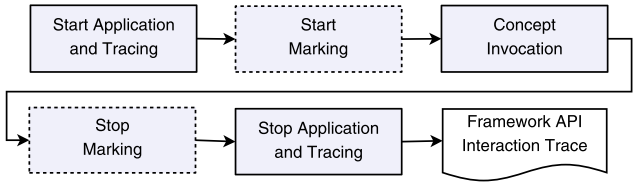


Fig. 4. Trace collection and marking (Step 3 of the FUDA process.

4.3 Selection of Sample Application(s) and Execution Scenarios

The user must identify one or more sample applications that execute the desired concept. Object-oriented frameworks often come with sample applications from which developers can learn frameworks' APIs. Sample applications can also be found using open-source code repositories (e.g., www.sourceforge.net) and code search engines (e.g., www.google.com/codesearch).

Further, the user needs to select two or more execution scenarios, each invoking an instance of the concept. Normally, two scenarios are needed, as shown in Section 6. FUDA generates best results if one or more of the following conditions are satisfied: 1) The scenarios are concept-focused, i.e., the majority of the executed instructions are part of the concept, 2) the scenarios execute the desired concept together with other concepts, but the user can mark the invocations of the desired concept at runtime, and 3) each scenario invokes the desired concept in a different context, i.e., the other executed concepts differ across the scenarios. A single application may already satisfy the third condition. For example, a single application implementing a context menu in two different views, such as table and tree, would suffice. Because FUDA works by intersecting traces of different executions (cf. Section 4.5.3), the more the contexts differ across the executions, 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 to minimize false negatives. For example, if a context menu with a separator is desired, scenarios that contain separators should be selected.

4.4 Trace Collection and Marking

Fig. 4 illustrates the process of trace collection and marking. In this step, the user executes the sample application(s) under a *tracer* and invokes the concept of interest according to the scenarios selected in the previous step. In order to understand how an application is using a framework API, the interactions between the application and the API need to be analyzed. To define the framework boundary for the

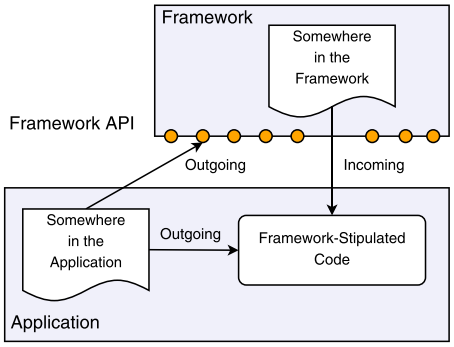


Fig. 5. Incoming and outgoing calls.

tracer, the user needs to specify the package(s) in which the framework resides—e.g., `org.eclipse.jface.*` for the context menu—and the package(s) in which the application resides. The tracer then logs all calls that occur at the boundary between the application and the framework, which results in a *framework API interaction trace*, or *API trace* for short.

To pinpoint the location of the concept execution in the trace, the user informs the tracer of the moments right before and after the invocation of the concept (dotted rectangles in Fig. 4). This technique is called *marking* [14] and will improve the template extraction results (cf. Section 6). For the context menu, marking amounts to instructing the tracer right before opening the menu to mark subsequent events and instructing it to stop marking right after the menu is open. However, the marking step is optional: If the moments before and after concept invocation cannot be pinpointed, FUDA considers the whole trace as marked.

The API trace consists of *API interaction events*, which are runtime events corresponding to method or constructor calls executed at the boundary between the framework and application code. From the viewpoint of application, each event has one of two directions (Fig. 5): 1) 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, and 2) an event is *incoming* if the call is made from within the framework code and the target is framework-stipulated and is implemented in the application code.

The complete API trace produced by running `SampleView` from Fig. 1 and invoking its context menu is shown in Fig. 6. Events are denoted as $D\ O{:}n(P){:}R$, where D represents the direction of the event, with "$\rightarrow$" for incoming and "$\leftarrow$" for outgoing events, O is the target object's ID or "null" for constructor and static method calls, n represents the fully qualified name 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 is `void`. For brevity, the package prefix `org.eclipse` was removed from n for all JFace events.

Most of the events in Fig. 6 can easily be traced back to their corresponding code lines in Fig. 1, e.g., e_1 corresponds to line l. 191. The calls in lines l. 190, l. 209, and l. 210 are not traced because they reside in `eclipse.ui`, which is not part of JFace. Indentation denotes *event nesting*. For example, events e_{11}-e_{16} were generated in the control flow

e_1 ←null:WelcomeWindow.<init>():1
e_2 ←1:WelcomeWindow.open():2
e_3 →1:jface.window.Window.createContents(3):3
e_4 ←1:WelcomeWindow.getShell():3
▲e_5 ←null:jface.viewers.TreeViewer.<init>(4,5):6
▲e_6 ←null:SampleView$ViewContentProvider.<init>(7):8
▲e_7 ←6:jface.viewers.TreeViewer.setContentProvider(8):V
▲e_8 ←null:SampleView$ViewLabelProvider.<init>(7):9
▲e_9 ←6:jface.viewers.TreeViewer.setLabelProvider(9):V
▲e_{10} ←6:jface.viewers.TreeViewer.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:SampleView$ViewContentProvider.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:SampleView$2.<init>(7):17
▲●e_{18} ←17:jface.action.Action.setText(18):V
▲●e_{19} ←17:jface.action.Action.setToolTipText(19):V
▲●e_{20} ←null:SampleView$3.<init>(7):21
▲●e_{21} ←21:jface.action.Action.setText(22):V
▲●e_{22} ←21:jface.action.Action.setToolTipText(23):V
▲●e_{23} ←null:jface.action.MenuManager.<init>(24):25
▲●e_{24} ←25:jface.action.MenuManager.setRemoveAllWhenShown(26):V
▲●e_{25} ←null:SampleView$1.<init>(7):27
▲●e_{26} ←25:jface.action.MenuManager.addMenuListener(27):V
▲●e_{27} ←6:jface.viewers.TreeViewer.getControl():28
▲●e_{28} ←25:jface.action.MenuManager.createContextMenu(28):29
▲●e_{29} ←6:jface.viewers.TreeViewer.getControl():28
▲●e_{30} ←6:jface.viewers.TreeViewer.getControl():28
●e_{31} **→27:jface.action.IMenuListener.menuAboutToShow(25):V**
●e_{32} **←25:jface.action.IMenuManager.add(17):V**
●e_{33} **←25:jface.action.IMenuManager.add(21):V**
●e_{34} **←null:jface.action.Separator.<init>():30**
●e_{35} **←25:jface.action.IMenuManager.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.IMenuManager.add(34):V**
▲e_{40} →8:jface.viewers.IContentProvider.inputChanged(6,10):V
▲e_{41} →8:jface.viewers.IContentProvider.dispose():V
e_{42} ←1:WelcomeWindow.close():35

Fig. 6. Framework API interaction trace.

of event e_{10}. Anonymous classes are denoted by numbers separated from their host classes by $, e.g., e_{17} constructs `action1` (l. 220). The events in bold face are those that were marked by informing the tracer about the moments just before and after the context menu was invoked. These events were generated by the callback to `menuAboutToShow()` (l. 205), which is called by JFace when a menu is being opened. That method calls `fillContextMenu()` (l. 212), which generates the nested events e_{32}-e_{39}.

4.5 Automated Trace Processing

The last step is to call an analyzer that takes the collected traces as input and generates a template. The analyzer performs four automatic substeps that are described below.

4.5.1 API Trace Slicing

The marked trace region (bold face) in Fig. 6 contains events that implement the context menu and potentially some other, irrelevant events. However, if the goal is to understand the complete life cycle of the concept, we also need to consider calls related to the initialization and clean up of the objects implementing the concept. These calls may not be part of the marked region. For example, the events e_{17}-e_{22} create and initialize the context menu's actions, but they are not in the marked region. Likewise, the marked region may miss cleanup events, such as object deregistration.

We apply *API trace slicing* to determine the relevant unmarked calls in the input trace. The heuristic behind this technique is that two calls are relevant if they share at least

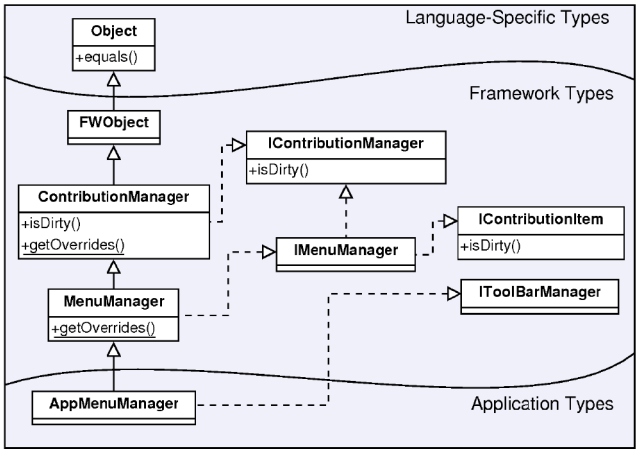


Fig. 7. Boundaries of application, framework, and language-specific types for menu-related classes.

one object as their target, parameter, or return objects. The motivation behind this heuristic is further discussed in Section 8.3. The precise definition of API trace slicing is based on the *object-connectedness* of two events.

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

Based on this definition, we define *object-relatedness* as the transitive closure of object-connectedness. Then, we define a *trace slice* as the portion of the input trace consisting of all the marked events and the unmarked events that are object-related to the marked events. In Fig. 6, the unmarked events that are object-related to the marked ones are identified by ▲. For example, e_5 is object-connected to e_7 through the object with ID 6, and e_7 is object-connected to e_{36} through object 8. Consequently, e_5 is object-related to the marked event e_{36} and thus part of the slice. Slicing eliminates the steps implementing the Welcome window (e_2-e_4, e_{42}), which are unrelated to the context menu. API trace slicing is an approximation of the actual data dependencies between API calls. Nevertheless, the approximation works well for real concepts and framework APIs, as shown in our evaluations in Section 6.

4.5.2 Event Generalization

The next processing step attempts to find commonalities among the sliced traces. To this end, we first abstract any application-specific elements from the events. The abstraction is achieved using *event generalization*, a static analysis that replaces application-specific names of events with appropriate framework API names, based on the application and API type hierarchy. For example, the fully qualified name of e_8 in Fig. 6, i.e., `SampleView$ViewLabelProvider.<init>`, is application-specific and event generalization replaces it with `jface.viewers.LabelProvider.<init>`. Event generalization treats calls to instance methods, constructors, and static methods differently. We will explain it using Fig. 7. The figure shows the menu-related classes in JFace.

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 implemented in many subclasses, it conceptually belongs to `Object`. A method may have multiple topmost types. For example, generalization of a call to `AppMenuManager.isDirty()` identifies both `IContributionManager` and `IContributionItem` as the topmost types because both interfaces declare the method.

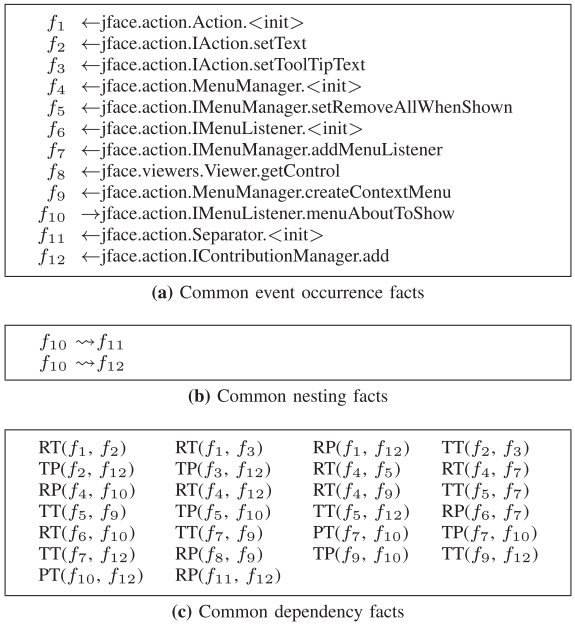
Constructors. An application class may specialize many framework and application-specific types. For constructor calls, the procedure aims at minimum generality and selects all framework-defined supertypes of the target application class that are located at the bottom framework borderline of the type hierarchy (see Fig. 7). For example, for a call to the constructor of `AppMenuManager`, the procedure identifies `MenuManager` and `IToolBarManager` as the generalized framework superclass and interface. The rationale behind minimum generalization for constructors is that selecting the topmost types, even if only the topmost *framework* types, would lose too much information. For example, assuming that all framework classes are derived from `FWObject`, the latter 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. For example, in Fig. 7, both `MenuManager` and `ContributionManager` declare the `getOverrides()` static method. Depending on which class is used statically, a different method is really being used. Thus, the procedure searches the type hierarchy of the application class being instantiated and returns the first type that declares the method.

4.5.3 Common Facts Extraction

The goal of this step is to identify the facts that are common among all input generalized traces. First, this step extracts three kinds of facts from each generalized trace: *event occurrence facts*, *event nesting facts*, and *event dependency facts*. The first kind of facts represents the occurrence of interaction events in the generalized trace, while the other two indicate the presence of certain relationships among events. *Common facts* are then computed as intersections of the extracted fact sets across all generalized traces. Fig. 8 illustrates these three kinds of common facts for the context menu. These facts were extracted from two generalized traces, one obtained from `SampleView` (Fig. 1) and the other one obtained from the `Console` application.

Event occurrence facts. These facts, called *event facts* for short, indicate the names of the methods and constructors that were called at the application-framework boundary and the corresponding call directions (Fig. 8a). 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 name but different parameter or return types or numbers of parameters are likely to be conceptually equivalent within an API. An event fact $Dt.n$, where t is a type name, is extracted from a generalized trace iff the trace contains one or more events $D\,O_i{:}[\ldots, t, \ldots].n{:}R_i$, where O_i is any object ID or "null" and R_i is any object ID or "V." We say that the events *match* such an event fact. For example, f_2 is extracted from the generalized trace because some of its events, e_{18} or e_{21} (see



(a) Common event occurrence facts

(b) Common nesting facts

(c) Common dependency facts

Fig. 8. Common facts extracted from generalized traces for `SampleView` and `Console` example applications.

Fig. 6), match it. All the events in Fig. 6 that match the common event facts in Fig. 8a are marked by •. The remaining events are effectively filtered out since they are unique to this trace.

Event nesting facts. Nesting facts record the calling context for outgoing calls (Fig. 8b). A nesting fact $f_i \rightsquigarrow f_j$, where f_i and f_j are event facts, is produced whenever the generalized trace contains two events e_k and e_l such that 1) e_k and e_l match f_i and f_j, respectively, 2) e_l is outgoing, and 3) e_l is directly nested in e_k in the trace.

Event dependency facts. This type of facts indicates call sequence and object passing patterns. There are nine types of dependency facts: target-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). A target-target dependency fact $\mathrm{TT}(f_i, f_j)$, where f_i and f_j are event facts, is produced whenever the generalized trace contains two events e_k and e_l such that 1) e_k and e_l match f_i and f_j, respectively, 2) e_k precedes e_l in the trace, and 3) both e_k and e_l have the same object as target. We obtain analogous definitions for the remaining dependency fact types 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 fact type is $\mathrm{RP}(f_i, f_j)$. 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.

4.5.4 Template Generation

The final automated substep is to generate the concept-implementation template. The common facts extracted from the generalized traces determine the overall structure of the template. This section sketches the main steps of the template generation algorithms; we refer the reader to [13] for further details. The template generation procedure includes the following steps.

Create classes. The first step in generating a template is to identify its constituent classes. The classes are identified using the common incoming method calls and outgoing constructor calls, as indicated by the set of common event facts. An incoming method call implies that the method should be implemented in an application class. An outgoing constructor call happens when an application class extends or implements one of the framework classes or interfaces. If two incoming method calls have the same target, then it means that they belong to the same class. Thus, a class is created for each group of incoming method calls that are related by TT dependencies in the set of common dependency facts. The corresponding constructor calls are assigned using RT dependencies. For example, the class in l. 9 (Fig. 2) is created for the fact f_{10}, which does not participate in any TT dependencies and thus forms its own group. The corresponding constructor call is f_6, due to $\mathrm{RT}(f_6, f_{10})$. The remaining unassigned constructor calls for abstract classes or interfaces, which occur when instantiating anonymous classes, are grouped through RR dependencies and a class is created for each such group. For example, the class in l. 9 is created for f_1, a call to the constructor of the abstract class `Action`.

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 is created for f_{10}. A constructor is created in a class if nesting facts whose source is any of the constructor calls assigned to that class are present.

Create statements. Outgoing calls are placed in method bodies based on the common nesting facts. For example, the nesting fact $f_{10} \rightarrow f_{12}$ (Fig. 8b) places the call in l. 12 (Fig. 2). Moreover, a class named `SomeClass` (l. 17), containing a method named `someMethod` (l. 18), is created to host the outgoing calls for which the nesting facts do not specify any calling contexts. For example, the set of nesting facts in Fig. 8b does not specify any calling contexts for the outgoing event facts f_1-f_9 and, thus, all of them are placed in `SomeClass.someMethod()`.

After specifying the calls that should go into the body of each method, the generalized traces are consulted to see whether the calls are repeated in a given calling context in all or some of the input traces. The calls are then commented with the terms `REPEAT` or `MAY REPEAT`, respectively. For example, the call to f_{12} is marked as `REPEAT` because f_{12} was called multiple times in every trace within the control flow of `menuAboutToShow()`. Finally, within each method, calls are sorted in an order determined by the dependency facts. In particular, the call order is obtained as a topological sort on the graph with event facts as nodes and the dependency facts as directed edges. As multiple calls to a given method are collapsed in a single event fact, dependency facts may form cycles, in which case only a subset of the calls can be sorted. The calls that cannot be sorted according to the dependencies are listed in an arbitrary order and the user is warned by a comment in the generated template.

Identify supertypes. Superclass and interfaces for each class (except `SomeClass`) are determined by constructing a

type hierarchy of target types of incoming method and constructor calls assigned to that class. The leaves of this type hierarchy identify the supertypes for that class.

Generate class and variable names. Each newly created class (except `SomeClass`) is named by prepending `App` to its superclass name or one of its interface names if no superclass is present. The method signature declared in the framework API is used to identify the return type and parameters of each method (except `someMethod()`). If there are several signatures for the same method name (e.g., different parameters), alternatives are shown using the "||" notation, as described in Section 3. Finally, constructor calls and method calls whose return types are not void are made initializers of variable declarations. Variables are then given names that are the same as their types, but in lower case, e.g., `appAction` in l. 21 and `menu` in l. 28.

Propagate variables. This step aims at showing how objects are passed among the program statements in the template. Using a graph in which program statements are nodes and the dependency facts (except RR) are directed edges among them, the variables defined in the statements in the previous step are propagated to their successor nodes. For example, `appAction` is passed as a parameter to `add` in l. 12 because of $\mathrm{RP}(f_1, f_{12})$. The notation "||" is used to illustrate different argument types passed to a method or constructor call, e.g., in l. 12. Parameter objects of framework-stipulated types that were not returned by any other calls are provided by dummy declarations as in l. 19.

Identify imports. This last step removes package names from the fully qualified names of types and creates the list of package imports.

5 FUDA Implementation

We implemented FUDA as a tool for Java consisting of two parts: *FUDA Tracer* and *FUDA Analyzer*. The FUDA Tracer is responsible for assisting the developer in collecting and marking the traces (Step 3 in Fig. 3). The FUDA Analyzer implements the automated trace analysis and template generation (Step 4). A demonstration of these tools is available online [15].

FUDA Tracer. The current implementation of FUDA Tracer instruments applications using aspects written in *AspectJ* [16], an aspect-oriented programming language. The aspects intercept both 1) calls from application methods to framework methods and 2) callbacks from framework methods to application methods. To identify the boundary between the application and the framework API, the user must specify the packages in which the framework of interest and the sample application reside. The tracer also provides a GUI allowing users to instruct the tool—by a push of a button—to start and to stop marking the events that the tool records.

The FUDA Tracer uses two alternative aspect weavers: *AJEER*[1] and *MAJOR* [17]. The AJEER plug-in is used to instrument code inside the Eclipse platform; however, this weaver cannot be used for frameworks that are part of the standard Java class library, such as Java Swing. On the other hand, the aspect weaver MAJOR is designed to weave aspects into the standard Java class library, but not into the Eclipse plug-ins. Thus, we use MAJOR for frameworks that are part of the standard Java class library.

FUDA Analyzer. FUDA Analyzer implements the trace analysis and template generation process (Section 4.5) as an Eclipse plug-in. The analyzer accepts the traces collected by the FUDA Tracer for a given concept and outputs the resulting template.

6 Quality of FUDA Templates

This section presents the first study, which assesses the quality of FUDA-generated templates for realistic framework-provided concepts.

6.1 Experiment Objectives

We designed the first experiment to answer the following research questions:

1. What is the quality of FUDA-generated templates in terms of precision and recall?
2. Would using more than two traces significantly increase precision and recall?
3. What is the impact of slicing on precision and recall?

To make FUDA more practical, we aim at keeping the number of traces per concept as small as possible. Therefore, the experiment gives results for using one, two, and three traces, each collected from a different application.

6.2 Experimental Setup

6.2.1 Selection of Frameworks

The evaluation uses five GUI frameworks (Tables 1 and 2): *Eclipse*,[2] *JFace*,[3] *Graphical Editing Framework*,[4] (*GEF*) *Java2D*,[5] and *Java Swing*.[6] These frameworks are widely used and sample applications for them are readily available in open source repositories.

6.2.2 Selection of Concepts

Tables 1 and 2 present the concepts used for this evaluation. We selected the Eclipse and JFace concepts, except Focus, based on our prior familiarity with the frameworks. We sampled the other concept-defining queries from developer forums of the respective frameworks [18], [19], [20], [21]. We selected 14 concepts in total and applied FUDA to them. After the application, we were able to characterize the concepts as follows:

- *Slicing.* Slicing is applicable to only some concepts. We do not use slicing if the invoking scenarios involve only the desired concept and span its entire lifecycle, e.g., opening and closing a view. Tree Viewer and Table Viewer in Table 1 and Text Editor in Table 2 are examples of such concepts. Template generation uses the full traces for these concepts. Further, the user may be interested only in the events that occurred when the concept was invoked.

1. http://ajeer.sourceforge.net/.

2. http://www.eclipse.org/.
3. http://wiki.eclipse.org/index.php/JFace.
4. http://www.eclipse.org/gef/.
5. http://java.sun.com/products/java-media/2D/.
6. http://java.sun.com/javase/6/docs/technotes/guides/swing/.

TABLE 1
Template Quality Evaluation for JFace, Eclipse, and GEF Concepts (* Indicates Concepts from Developer Forums)

Frameworks	Concepts	Defining Questions	Properties				Sample Applications			Precision and Recall Results									
										No Slicing					With Slicing				
			Slicing	Frequent	Simple	Variable	Numbers of Samples	Names	Sources	G	I	M	P	R	G	I	M	P	R
JFace	Context Menu	How to implement a context menu in a view?	X	X	X	X	1	Console	Eclipse UI	69	40 (56)	1 (1)	42 (19)	97 (93)	66	37 (53)	1 (1)	**44 (20)**	**97 (93)**
							2	Tree View	Eclipse Wizard	15	0 (4)	1 (1)	100 (73)	94 (92)	15	0 (4)	1 (1)	**100 (73)**	**94 (92)**
							3	ANT View	Eclipse UI	15	0 (4)	1 (1)	100 (73)	94 (92)	15	0 (4)	1 (1)	**100 (73)**	**94 (92)**
	Toolbar Button	How to add a button to a view's toolbar?	X	X	X	X	1	Package Explorer	Eclipse JDT	289	248 (285)	3 (3)	14 (1)	93 (57)	263	230 (259)	3 (3)	**13 (2)**	**92 (57)**
							2	Crosscutting Comparison	AJDT	18	5 (14)	3 (3)	72 (22)	81 (57)	13	1 (9)	3 (3)	**92 (31)**	**80 (57)**
							3	Debug View	Eclipse UI	17	4 (13)	3 (3)	76 (24)	81 (57)	13	1 (9)	3 (3)	**92 (31)**	**80 (57)**
	Content Assist	How to develop a content assistant in a text editor?	X	–	–	X	1	JSP Editor	Eclipse WTP	299	244 (283)	1 (1)	18 (5)	98 (94)	233	178 (217)	1 (1)	**24 (7)**	**98 (94)**
							2	Java Editor	Eclipse JDT	46	27 (30)	1 (1)	41 (35)	95 (94)	32	13 (16)	1 (1)	**59 (50)**	**95 (94)**
							3	HTML Editor	Eclipse WTP	38	22 (24)	3 (3)	42 (37)	84 (82)	24	8 (10)	3 (3)	**67 (58)**	**84 (82)**
Eclipse	Table Viewer	How to develop a table viewer?	–	X	–	X	1	LDAP Connections	eclipse-plugins	379	312 (363)	1 (1)	**18 (4)**	**99 (94)**	–	–	–	–	–
							2	Table View	Eclipse Wizard	39	0 (23)	1 (1)	**100 (41)**	**98 (94)**	–	–	–	–	–
							3	Editor List	eclipse-plugins	34	0 (18)	1 (1)	**100 (47)**	**97 (94)**	–	–	–	–	–
	Tree Viewer	How to develop a tree viewer?	–	X	–	X	1	LDAP Browser	eclipse-plugins	208	147 (192)	2 (2)	**29 (8)**	**97 (89)**	–	–	–	–	–
							2	Tree View	Eclipse Wizard	45	0 (29)	2 (2)	**100 (36)**	**96 (89)**	–	–	–	–	–
							3	Concern Mapper	Google Search	40	0 (24)	2 (2)	**100 (40)**	**95 (89)**	–	–	–	–	–
	Navigate	How to create a tree viewer with tree navigation (i.e., *Go Home*, *Go Back* and *Go Into*) buttons in its toolbar?	X	–	–	X	1	SVN Repository	Subclipse	238	198 (218)	0 (0)	17 (8)	100 (100)	174	139 (154)	0 (0)	**20 (11)**	**100 (100)**
							2	KTreeMap	SourceForge	40	10 (20)	0 (0)	75 (50)	100 (100)	38	8 (18)	0 (0)	**79 (53)**	**100 (100)**
							3	AST View	Eclipse JDT	37	8 (17)	0 (0)	78 (54)	100 (100)	36	7 (16)	0 (0)	**81 (56)**	**100 (100)**
	Focus*	What events happen by clicking on a view's titlebar?	–	X	X	–	1	LDAP Browser	eclipse-plugins	4	0 (0)	0 (0)	**100 (100)**	**100 (100)**	–	–	–	–	–
							2	Editor List	eclipse-plugins	4	0 (0)	0 (0)	**100 (100)**	**100 (100)**	–	–	–	–	–
							3	Table View	Eclipse Wizard	4	0 (0)	0 (0)	**100 (100)**	**100 (100)**	–	–	–	–	–
GEF	Select*	What events happen by clicking on a figure?	–	X	X	–	1	Flow	GEF Examples	16	9 (12)	0 (0)	**44 (25)**	**100 (100)**	–	–	–	–	–
							2	Shapes	GEF Examples	7	0 (3)	0 (0)	**100 (57)**	**100 (100)**	–	–	–	–	–
							3	Logic	GEF Examples	7	0 (3)	0 (0)	**100 (57)**	**100 (100)**	–	–	–	–	–
	Figure*	How to draw a figure in a GEF editor?	X	X	–	X	1	Flow	GEF Examples	145	74 (137)	0 (0)	49 (6)	100 (100)	130	59 (122)	0 (0)	**55 (6)**	**100 (100)**
							2	Shapes	GEF Examples	83	25 (75)	0 (0)	70 (10)	100 (100)	68	10 (60)	0 (0)	**85 (12)**	**100 (100)**
							3	Logic	GEF Examples	80	23 (72)	0 (0)	71 (10)	100 (100)	66	9 (58)	0 (0)	**86 (12)**	**100 (100)**
	Connection*	How to draw a connection between two figures?	X	X	–	X	1	Flow	GEF Examples	155	74 (145)	0 (0)	52 (6)	100 (100)	140	59 (130)	0 (0)	**58 (7)**	**100 (100)**
							2	Shapes	GEF Examples	91	25 (81)	0 (0)	73 (11)	100 (100)	76	10 (66)	0 (0)	**87 (13)**	**100 (100)**
							3	Logic	GEF Examples	88	23 (78)	0 (0)	74 (11)	100 (100)	74	9 (64)	0 (0)	**88 (14)**	**100 (100)**

G: Template sizes; I: Numbers of false positives; M: Numbers of false negatives; P: Precisions; R: Recalls.

In this case, we use the marked region. Focus and Select in Table 1 are examples of such concept invocations. The life cycles of the remaining concepts span beyond the marked trace region and we apply slicing to them (see the concepts with "X" in the Slicing column).

TABLE 2
Template Quality Evaluation for Java2D and Java Swing Concepts (* Indicates Concepts from Developer Forums)

Frameworks	Concepts	Defining Questions	Properties				Sample Applications			Precision and Recall Results										
										No Slicing					With Slicing					
			Slicing	Frequent	Simple	Variable	Numbers of Samples	Names	Sources	G	I	M	P	R	G	I	M	P	R
Java 2D	Moving Shapes*	How to draw shapes and let the user drag them?	X	X	X	X	1	GeoSoft	Google Search	112	80 (101)	0 (0)	29 (10)	100 (100)	95	64 (84)	0 (0)	**33 (12)**	**100 (100)**
							2	GTEditor	Google Code	27	7 (16)	2 (2)	74 (41)	91 (85)	20	3 (9)	2 (2)	**85 (55)**	**89 (85)**
							3	JHotDraw	SourceForge	27	7 (16)	2 (2)	74 (41)	91 (85)	20	3 (9)	2 (2)	**85 (55)**	**89 (85)**
	Circle Drawing*	How to draw a red circle on a black background?	X	X	X	X	1	TerpPaint	SourceForge	56	48 (53)	0 (0)	14 (5)	100 (100)	36	28 (33)	0 (0)	**22 (8)**	**100 (100)**
							2	Scribble	Google Search	11	4 (8)	0 (0)	64 (27)	100 (100)	9	2 (6)	0 (0)	**78 (33)**	**100 (100)**
							3	JHotDraw	SourceForge	9	4 (8)	2 (2)	56 (11)	71 (33)	7	2 (6)	2 (2)	**71 (14)**	**71 (33)**
Java Swing	Drag-n-Drop*	How to drag-n-drop an item in a tree?	X	X	-	X	1	Java2s DnD	Google Search	94	21 (76)	0 (0)	78 (19)	100 (100)	90	18 (72)	0 (0)	**80 (20)**	**100 (100)**
							2	Swing Demo	Google Search	77	8 (59)	0 (0)	90 (23)	100 (100)	76	7 (58)	0 (0)	**91 (24)**	**100 (100)**
							3	TV Browser	SourceForge	54	7 (36)	0 (0)	87 (33)	100 (100)	54	7 (36)	0 (0)	**87 (33)**	**100 (100)**
	Text Editor*	How to develop a simple text editor?	-	X	-	X	1	Edas Texter	SourceForge	59	4 (37)	0 (0)	**93 (37)**	**100 (100)**	–	–	–	–	–
							2	nText	SourceForge	47	3 (25)	0 (0)	**94 (47)**	**100 (100)**	–	–	–	–	–
							3	DrJava	SourceForge	39	1 (17)	0 (0)	**97 (56)**	**100 (100)**	–	–	–	–	–

G: Template sizes; I: Numbers of false positives; M: Numbers of false negatives; P: Precisions; R: Recalls.

- *Frequency.* A concept is either *frequent* or *rare* among the existing applications of the framework. FUDA is readily applicable to frequent concepts as finding sample applications for them is likely easy; however, it is also applicable to rare concepts, provided that the user has already identified one or two applications with appropriate execution scenarios. Concepts that may seem rare at first might not be rare after all. For example, the particular choice of red and black in Circle Drawing may be rare, but setting background and figure colors is not.
- *Complexity.* A concept is either *simple* or *complex* in terms of implementation complexity measured as template size, i.e., the number of implementation steps. We used 20 as the upper bound for the number of steps for simple concepts. In our experiment, simple concepts have between 4 and 20 steps; complex concepts have between 32 and 76 steps.
- *Variability.* A concept is either *variable*, if it has optional implementation steps, or *fixed*, if it has only a fixed set of steps. The majority of the considered concepts are variable. For example, a context menu may or may not include a separator. When applying FUDA, the user must be aware that optional steps will be lost if they are not present in both traces. Thus, if the user is interested in context menus with separators, both traces must involve such menus.

6.2.3 Selection of Sample Applications and Execution Scenarios

We chose three sample applications implementing the desired concept in three different contexts and, for each of those applications, we designed one execution scenario. 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, we designed the scenarios relying on documentation or prior familiarity with the concept or by playing with sample applications.

Tables 1 and 2 present the sample applications and their sources.

Selection of the applications involved the following strategies:

1. reliance on prior familiarity with a given application (mostly for Eclipse and JFace concepts, which were part of a larger familiar environment such as Java Development Tools (JDT) or Eclipse Web Tools Platform (WTP) or were generated by Eclipse wizards),
2. browsing and running the standard examples of the framework (mostly for GEF concepts),
3. searching or browsing in online application repositories (e.g., 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
4. tips by others (e.g., 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 the authors' prior familiarity to up to an hour of searching and browsing on eclipse-plugins.org or SourceForge.net.

6.2.4 Trace Collection

We used the FUDA Tracer (cf. Section 5) to collect the API traces. API tracing is efficient because 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. For example, tracing boundary calls for GEF was almost unnoticeable when using GEF applications. However, the applications ran two to three times slower when boundary calls of Eclipse were 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 1.6 MHz processor, 1 GB of RAM, and Windows XP.

6.2.5 Template Generation

We used the FUDA Analyzer (cf. Section 5) to generate the implementation templates. The running time for this step was anywhere from a few seconds up to six minutes. The main bottleneck of the process was the number of dependency facts generated. The largest number of dependency facts generated was about 3.7 million for Content Assist in HTML Editor, which resulted in the longest processing time of six minutes. However, performance optimization has not been a goal in our implementation.

6.3 Analysis Procedure

The analysis procedure includes both quantitative and qualitative analyses. The quantitative data are the main source for answering the research questions formulated in Section 6.1, while qualitative analysis provides a deeper insight into the quantitative results.

6.3.1 Quantitative Analysis Procedure

A reference is required to calculate the precisions and recalls of the generated templates. For this purpose, we created a *mandatory* and an *optional reference template* for each concept. Mandatory reference templates represent the set of mandatory implementation steps, i.e., the ones that are necessary to the instantiation of a concept: If the step is removed, the concept does not work as expected. For example, a context menu cannot be realized without calling the method `createContextMenu()` (l. 28 in Fig. 2). Optional reference templates additionally include steps that are not required but that are relevant to the concept and were present in the sample applications. For example, Context Menu's optional reference template includes calls to create and register separators.

We created the reference templates carefully to minimize threats to the validity of the results. To ensure their correctness, we consulted online documentation (third-party articles and solutions posted in forums), manually inspected the sample applications, and also used the templates to develop sample implementations of the concepts. 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. We also compared the reference templates against the generated ones to identify optional features present in the sample applications. Each nonmandatory 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.

The calculation of precision and recall was based on counting the implementation steps contained in a template and comparing them against the reference templates. The considered steps were superclass declarations, implement declarations (i.e., declaring that a class implements an interface), method implementations, method calls, and constructor calls. These steps are the main elements of a template. Call sequence and parameter passing patterns were considered supplementary information that make templates more readable and were not used in the evaluation. The calculation of precision and recall involved these three numbers:

- G: The number of all implementation steps in the $g$enerated template.
- I: The number of steps that are $i$ncorrectly present in the generated template, but absent in the reference template (i.e., false positives).
- M: The number of steps that are present in the reference template but $m$issing in the generated template (i.e., false negatives).

Precision (P) was calculated as $P = (G - I)/G$, and *recall* (R) is calculated as $R = (G - I)/(G - I + M)$. To reduce bias, we calculated the ranges of values possible for the precisions and recalls of generated templates. Their lower bounds were determined by using the mandatory reference templates, i.e., the optional steps in the generated templates were considered false positives. The upper bounds were calculated with the help of optional reference templates, i.e., the optional steps in the generated templates were not counted as false positives.

6.3.2 Qualitative Analysis Procedure

We inspected the generated templates, the sample applications used to generate them, and various intermediate results (e.g., collected traces, sliced traces, and common facts) to gain insights into the causes of false positives and false negatives.

6.4 Analysis Results

6.4.1 Quantitative Results

Tables 1 and 2 illustrate the results of the quantitative analysis for when one, two, and three sample applications are used. Given three applications per concept, we randomly selected one to determine the result for one application. For example, Context Menu has three sample applications: Console, Tree View, and ANT View. The row with "Numbers of Samples" of 1 gives the results for one application, which is Console. We then randomly selected the second application form the remaining two and computed the results for the first and second application. In our example, the row with "Numbers of Samples" of 2 gives the results for two applications, namely, Console and Tree View. Finally, we computed the results for all three applications (the row with "Numbers of Samples" of 3).

Tables 1 and 2 give I, M, P, and R numbers with respect to both mandatory and optional reference templates. The numbers with respect to mandatory reference templates are given in parentheses. They represent lower bounds, as optional steps may or may not have been desired by the user. Concepts to which no slicing applies have only one set of numbers under "No Slicing." For concepts to which slicing does apply, we give both the numbers for full traces

("No Slicing") and the numbers "With Slicing" in order to show the impact of slicing. The precision and recall numbers produced by the complete application of FUDA (i.e., with slicing, when applicable) are marked in bold.

The precision of the generated templates is much less with respect to mandatory reference templates than optional reference templates because these templates do not include any of the optional implementation steps, and thus we count all these optional steps in the generated templates as false positives. For example, the code in the mandatory reference template for Toolbar Button would produce an empty button, i.e., one without any icon, caption, and tooltip text; however, in practice, hardly any user is interested in the concept of an "empty toolbar button" as opposed to a "toolbar button with some caption and icon." Because the optional steps present in the generated templates were also present in sample applications, the precision and recall results for optional reference templates are more likely to adequately reflect the template quality as it will be experienced by most users. However, if a user had not been interested in some of the optional steps, the precision from that user's viewpoint would lie somewhere between the precision values for the mandatory reference templates and the values for the optional reference templates.

When only a single sample application is used, the sizes of the templates are larger than when two or three sample applications are used because FUDA uses the intersection among the traces. However, they are polluted with many false positives such that the precision for most of the concepts is around 50 percent or less. The recall is high, in the range of 92-100 percent, as no events are removed from the traces due to intersection. For the concepts Text Editor and Drag-n-Drop, because the sample application was simple and focused on the concept of interest, even a single application was enough to produce good results.

As we use the second sample application, the precisions of the templates significantly improve such that, except for the precision for Content Assist (59 percent) and Navigate (79 percent), all other precision and recall values are 80 percent or higher. However, when the third sample application is used, we often see that a few optional steps are being removed from the templates. Consequently, the resulting templates concentrate more on the set of mandatory implementation steps, without much improvement in precision and recall. For example, for the concepts Context Menu and Moving Shapes, no changes were observed in the results, while for the rest of the concepts there were only some slight changes. At the same time, the use of more sample applications increases the chance of false negatives, as different applications may use different instructions to implement the same functionality. For example, adding the second sample application for Moving Shapes and adding the third sample application for Content Assist and Circle Drawing introduced new false negatives.

The results show that slicing, where applicable, significantly improved the precision. Specifically, slicing eliminated between 7-42 percent, 13-80 percent, and 13-75 percent of false positives for one, two, and three sample applications, respectively—except for Context Menu and Drag-n-Drop, for which the applications were different enough, so that slicing had almost no effect.

These quantitative results answer our research questions (Section 6.1) as follows:

1. FUDA can generate templates with relatively few false positive and negatives. In particular, all but one generated templates achieved precision of at least 78 percent and recall of at least 89 percent. Further, more than half of the templates had precision of 90 percent or better and recall of 100 percent.
2. Using three traces instead of two did not bring any substantial improvements; in fact, it had even negative impact on recall in some cases. Thus, FUDA seems to perform best for two traces, collected in different contexts.

6.4.2 Qualitative Results

In general, false positives were more frequent than false negatives, particularly when just one trace was used. False positives were mainly due to similarities among the sample applications that extend beyond the concept of interest. For example, the single false positive for Toolbar Button with two sample applications was due to calls to `IShellProvider.getShell()`, a method frequently used in Eclipse views. False negatives resulted mainly from 1) necessary implementation steps that resided in packages other than the framework packages specified to the FUDA Tracer during the trace collection, or 2) the possibility of implementing the same concept in multiple ways. As an example of 1, the template generated for Context Menu was missing a mandatory call to `setMenu()` because the method was in the Eclipse framework, not in JFace. As an example of 2, the template generated for Moving Shapes with two and three sample applications had some false negatives because the applications used different instructions of the Java class library to change the location of a shape.

Our investigations revealed that slicing had a particularly positive impact on reducing false positives when the traces exercised more concepts in common than the desired one. For example, the example applications for the GEF Figure also shared other concepts such as Palette and Toolbar, and thus 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. Nevertheless, slicing is not beneficial when 1) sample applications share only the desired concept (e.g., for Menu and Drag-n-Drop, slicing had almost no impact when two and three sample applications were used), and 2) the user is interested in the whole trace (e.g., Table Viewer) or just the events in the marked region (e.g., Focus).

6.5 Threats to Validity

This section discusses the potential threats that may impact the validity of the experimental results.

6.5.1 Internal Validity

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 [22].

The main threat to internal validity is incorrect reference templates, which would impact the calculations of precisions and recalls. This threat was minimized by 1) using three sources of knowledge for all concepts: manual inspection of sample applications, consulting existing documentation, and testing the implementation steps in sample implementations; 2) having two of the authors independently check, in several iterations, the correctness of all the reference templates and the values calculated for precisions and recalls; and 3) reporting the ranges of values possible for precisions and recalls based on mandatory and optional reference templates.

Another threat to internal validity is related to problems that might have occurred during the process of trace collection, such as not triggering the marking at appropriate times. This issue was addressed by carefully designing and applying the execution scenarios.

Finally, the procedure of selecting one or two sample applications from three (cf. Section 6.4.1) can influence the results for one and two applications, creating the possibility of bias. We minimized this threat by selecting the first and the second application randomly.

6.5.2 External Validity

External validity relates to the extent to which the research questions capture the objectives of the research and the extent to which any conclusions can be generalized [22].

One potential threat to external validity is that the selection of frameworks for the evaluation might not have been representative of those used in realistic development. This threat is addressed by selecting five popular frameworks that are widely used in practice. However, because the selected frameworks are all GUI frameworks, it is still necessary to experiment with other kinds of frameworks.

Further, the concepts selected for the evaluation might not have been representative of those used in real-world development. We addressed this threat by including concepts from developer forums [18], [19], [20], [21].

The precisions and recalls are highly dependent on how the given concept is implemented in sample applications, and consequently, the way we selected the sample applications directly influences the results. We minimized this threat by following the same identification strategies that would be applied in practice (cf. Section 6.2.3).

6.5.3 Construct Validity

The test of construct validity questions whether the theoretical constructs are interpreted and measured correctly [22]. For 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. In particular, false positives and negatives depend on user's definition of the concept and thus are subjective to some extent. We minimized this threat by reporting numbers with respect to both the optional and mandatory reference templates.

6.5.4 Replicability

We 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 [15]. Consequently, it should be possible to replicate the experiment.

7 Usage of FUDA Templates

This section presents the results of a user experiment in which subjects used templates to implement framework-provided concepts.

7.1 Experiment Objectives

The goal of this experiment is to see whether FUDA templates are useful in practice. We ask skilled Java developers to implement framework-provided concepts with the aid of either templates or documentation and compare the effectiveness of the two aids. *The rationale behind this experiment is that if templates are at least as effective as documentation then they can serve as a substitute when no documentation is available.* This experiment attempts to answer the following research questions.

1. Are templates at least as effective as documentation in aiding the developers in concept implementations? We measure effectiveness in terms of *implementation time* and resulting code *correctness*.
2. What is the influence of a template's quality and its usage strategies on the quality of resulting implementations?

The only independent variable in this experiment is *documentation aid* with two values: *framework documentation* (*D*) and *implementation template* (*T*). Moreover, the two dependent variables that we study in this experiment are 1) the *time* to complete the assigned concept implementation task, and 2) the *functional correctness* of the implementation code. We measure correctness as a factor with three levels: *success* if the resulting implementation behaved as specified, *buggy* if the implementation did not perform as specified, and *failure* if the developer did not finish the implementation. For the first research question, we formulate the following null hypothesis:

> H_0: *There is no difference in time when implementing a concept with the help of templates or documentation.*

We claim that templates are more effective than documentation as concept-implementation aids. Thus, we formulate the following alternative hypothesis:

> H_1: *Use of templates reduces the time developers take to implement a concept compared to when they use documentation.*

To answer the second research question, we qualitatively analyzed developer feedback and the concept implementations, as explained in Section 7.4.

7.2 Experiment Setup

The experiment targets a context representative of situations where skilled Java programmers perform realistic concept implementation tasks for the first time on top of a nontrivial framework such as Eclipse.

Frameworks. We select Eclipse and JFace as the target frameworks because they are complex and widely used. We apply balancing to control for any differences between

TABLE 3
Assignment of Concept Implementation Tasks to Subjects (①and ②Represent the Order of Task Performance)

Moderate Subjects				Experienced Subjects			
Context Menu (Simple - Frequent)		Table Viewer (Complex - Frequent)		Navigate (Simple - Rare)		Content Assist (Complex - Rare)	
T	D	T	D	T	D	T	D
-	① S_{17}, S_{19}, S_{21}	② S_{17}, S_{19}, S_{21}	-	-	② S_1, S_4, S_6	① S_1, S_4, S_6	-
-	② S_{15}, S_{16}, S_{18}, S_{20}	① S_{15}, S_{16}, S_{18}, S_{20}	-	-	① S_2, S_3, S_5, S_7	② S_2, S_3, S_5, S_7	-
① S_{23}, S_{24}, S_{28}	-	-	② S_{23}, S_{24}, S_{28}	② S_9, S_{11}, S_{14}	-	-	① S_9, S_{11}, S_{14}
② S_{22}, S_{25}, S_{26}, S_{27}	-	-	① S_{22}, S_{25}, S_{26}, S_{27}	① S_8, S_{10}, S_{12}, S_{13}	-	-	② S_8, S_{10}, S_{12}, S_{13}

them; that is, for each framework, the number of tasks with treatment T is the same as the number of tasks with treatment D.

Concepts. Among the four concept characteristics mentioned in Section 6.2.2, frequency and complexity are most relevant for this experiment. We use frequency to classify subjects' experience with the target framework: Subjects are *experienced* if they have implemented the frequent concepts before and *moderate* otherwise. Complexity matters as it influences the implementation difficulty. Thus, we select four concepts from Section 6.2.2, each representing a different combination of complexity and frequency: Context Menu, Table Viewer, a simplified Navigate, and Content Assist, (see Table 3, the entire table will be explained shortly). The simplified Navigate does not require developers to implement a tree viewer (cf. Table 1).

While slicing and variability characteristics of concepts mainly influence the quality of the generated templates, we use in this experiment the templates that have been already generated in the previous study. Thus, the effects of these characteristics are limited in this experiment.

Target applications. Each concept has a *target application*, that is an Eclipse project in which the subjects are asked to implement that concept. For example, the target application for Content Assist is a simple text box without a content assistant. We keep the sizes of these projects minimal to help developers focus on implementing the assigned concepts instead of spending their time on investigating the target applications. Their sizes range between 10 (for Content Assist) and 186 lines of code (LOC) (for Navigate).

Implementation templates and sample applications. For each concept, we give subjects the template generated with the help of two sample applications in the previous study (cf. Section 6). We also give them these two sample applications as examples of actual implementations of the concept. The sizes of the sample applications varied between 1 KLOC (EditorList for Table Viewer) and 66 KLOC (Subclipse for Navigate).

Documentation. We use *standard framework documentation* sourced from the Eclipse Help, Eclipse Corner Articles,[7] or third-party Eclipse articles (web search). The documentation for each concept contained the code snippets required for the concept implementation. Whereas templates give the complete code in one piece, these snippets are distributed over multiple sections in the documentation. For example, for Context Menu, one section discusses how to create menu items and another section describes how to put those items together to create the menu. The length of these documents ranges between 5 pages (for Navigate) and 28 pages (for Content Assist). For each concept except Context Menu, we provide a second document with additional background information on the concept. The original document for Context Menu already has enough such information.

7.3 Experiment Procedure

We followed the procedure shown in Fig. 9.

Recruiting subjects. We sent an initial package to potential subjects. The package included 1) an *overview document* with a brief introduction to the experiment and the process that a subject would go through and 2) a *background questionnaire* that captured the subject's background and experience. We used the questionnaire to select suitable subjects. The recruiting criteria were Java proficiency. All recruited subjects had at least three years of Java programming experience and self-rated their Java programming skills at least 4 on a scale 1-5. Participation was voluntary and unpaid.

We recruited 28 subjects; we label them S_1-S_{28}. The subjects included three professionals (S_{14}-S_{15}, S_{27}) from three different software companies, 24 graduate students (S_1-S_{13}, S_{16}-S_{26}), and one senior undergraduate student (S_{28}). A total of 21 subjects had industry experience, ranging between 1 and 10 years.

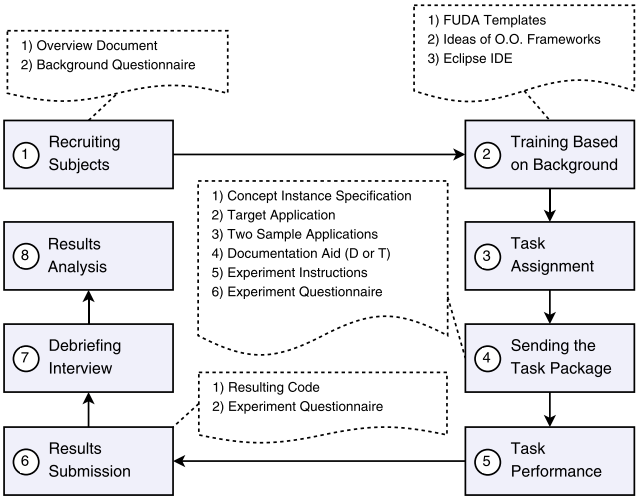


Fig. 9. Procedure of template usage evaluation.

7. http://www.eclipse.org/articles/.

Since the subjects had varying experience with the target frameworks (JFace and Eclipse), we blocked them into two groups: 1) *experienced subjects* (S_1-S_{14}) had previous experience with developing both frequent concepts (Context Menu and Table Viewer), but had not implemented the rare ones (Navigate and Content Assist), and 2) *moderate subjects* (S_{15}-S_{28}) had not previously implemented any of the four concepts of the experiment. We used the blocking to control for the experience with the target framework by balancing the assignment of treatments over the blocks (cf. Table 3).

Training based on background. We sent up to three different tutorials to each subject based on his responses to the background questionnaire. These tutorials included: 1) a brief document describing the *syntax and the semantics of FUDA templates*, 2) a short document introducing the *ideas of object-oriented software frameworks*, and 3) a short tutorial about how to use the *Eclipse IDE*.

In addition to above tutorials, each subject was given a personalized training session. During these sessions, we questioned subjects to become confident that they understood the tutorial contents. For example, we asked them to describe to us how a framework is used, how to compile and run projects in the Eclipse environment, or the meaning of special notations (e.g., "||") in a FUDA template.

Task assignment. To obtain more data points with fewer subjects, we assigned two concept implementation tasks to each subject (see Table 3). The assignment was random modulo the following constraints. First, we made sure that each subject was to implement the assigned concept for the first time. Second, the assignment had to balance treatments over the simple and complex concepts within each subject block. Further, as each subjects had to use a template for one concept and documentation for the other concept, we also balanced the treatment sequence within each block and within all subjects, i.e., half of the subjects first used documentation and then templates and the other half first used templates and then documentation.

Sending the task package. After task assignments, we sent a task package per concept to its corresponding subject. The task package contained the following items:

1. a *concept instance specification* with a description of the concept to be implemented,
2. the *target application* in which the subject was supposed to implement the given concept (cf. Section 7.2),
3. the *two sample applications* from which the template had been generated (cf. Section 7.2) to just provide subjects examples of actual implementations of the concepts,
4. either the *implementation template* or the *documentation* of the given concept,
5. a document representing a set of *instructions* that the subject must follow before and during the task performance, and
6. an *experiment questionnaire* that was used by the subject after the task completion to provide us feedback on the experiment such as implementation time, whether the provided documentation aid was useful, and how the templates could be improved.

Before the task performance, the subjects were instructed to read the tutorials and investigate the provided target applications. The intent of this rule was to ensure that the time recorded for task performance was the time spent on implementing the concept rather than investigating the target application. The subjects were not allowed to read the provided template or documentation and investigate the provided two sample applications before the task performance, however.

Task performance. Subjects performed the assigned tasks at their regular work place according to the instructions in the task package. In particular, we asked them to do the assigned tasks without interruptions and to record the actual time used for each task. During the implementation, the subjects were asked to use only the provided documentation aid (T or D), the two sample applications, and the framework-provided Javadoc documentation. In particular, they could not use Eclipse Help, Eclipse wizards, or search the web. The Javadoc documentation does not explain how to implement concepts, but only how to use a given framework-provided API element, such as an interface or a method.

Results submission. Immediately after performing each task, the subjects were instructed to fill the experiment questionnaire provided in the task package and return it, together with their concept implementation code, via e-mail.

Debriefing Interview. Upon submitting the results for both assigned tasks, each subject participated in a short debriefing interview. The main question in this interview was to understand the template usage strategy that the subject followed.

Results analysis. The quantitative assessment of the impact of the independent variable *documentation aid* on the dependent variable *implementation time* involved applying the parametric, one-tailed Student's t-test at a confidence level of 95 percent ($\alpha = 0.05$). To validate that the t-test can be used, we first applied the Kolmogorov-Smirnov test to verify normal distribution of samples. We also applied Levene's test to check for equality of variance in the samples. A p-value $> \alpha$ means that samples have equal variances.

In addition to confirming an effect as statistically significant, we also wanted to know its size. We computed the effect size by using the Cohen's d, which is defined as the difference between two means divided by the pooled standard deviation for those means. Since we used the difference between the T and the D groups, a negative value of Cohen's d corresponded to the T aid being more beneficial than the D aid. To interpret the Cohen's d values, Cohen suggested that the effect size is small for d around 0.2, medium for d around 0.5, and large for d of 0.8 and larger.

For the dependent variable *functional correctness of the implementation code*, we provide the outcomes (success, buggy, or fail) for the T and D groups. Since we had only six unsuccessful implementations, it did not make sense to perform statistical analyses.

The qualitative analyses involved careful examination of questionnaires and debriefing interviews and executing the submitted implementations to see if they had the correct functionality as described in the concept instance specification.

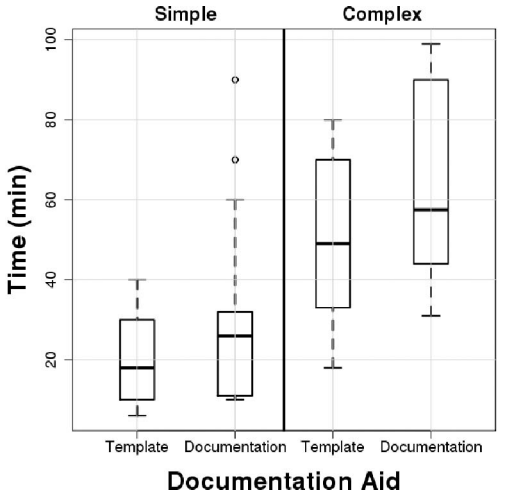


Fig. 10. Boxplot of concept implementation times.

7.4 Analysis Results

Quantitative analysis results. Fig. 10 gives a boxplot of the time spent for each implementation as a function of the documentation aid (T or D) and concept complexity. The boxplot considers complexity to give an idea of how this factor impacts time.

Table 4 gives the descriptive statistics and the results of statistical analyses. The first row in Table 4 gives the main analysis with documentation aid as the single independent variable and how it impacts implementation time. For comparison, rows two and three also analyze the impact of concept complexity and subject experience. The mean, standard deviation, minimum, median, and maximum values are given in minutes.

The analysis in row one indicates that the T group had better times than the D group. Cohen's d of 0.46 points to a medium effect size. We also successfully applied the Kolmogorov-Smirnov test and the Levene's test. Thus, we were able to apply the Student's t-test to evaluate the null hypothesis H_0. Because the observed t is smaller than the critical t and the p-value is lower than $\alpha = 0.05$, the application of the t-test allows us to reject the null hypothesis and instead accept the alternative hypothesis, which means that the implementation time is reduced statistically significantly by using templates instead of framework documentation. Our qualitative analyses suggest some likely reasons for why the T group performed better than the D group. We will discuss these reasons later in this section. Finally, the analyses in rows two and three indicate that the effect sizes for concept complexity and subject experience were larger than that for documentation aid.

Results of running code. Running and testing the code for all concept implementations revealed 1) three buggy implementations and one failure for the D group and 2) only two buggy implementations for the T group. In the D group, subjects S_{10} and S_{11} did not implement Content Assist as described in the concept instance specification, and subject S_{25} implemented a List Viewer instead of a Table Viewer. Furthermore, subject S_{23} decided to give up the implementation of Table Viewer after 90 minutes. In the T group, subject S_8 implemented Navigate with an additional button and S_{19} did not implement a toolbar for his Table Viewer as requested.

Subjects' comments on templates. All subjects except S_7 and S_{23} declared in their experiment questionnaires that the templates provided them with useful information. For example, S_1 stated: *"Yes, the templates were useful. The templates helped to limit the scope in which I needed to analyze the provided sample framework application. I only had to refer to a set of approx. three classes in each application. In addition, the template provided useful boiler-plate code which I could use as a starting point."* As many as 24 out of 28 subjects used the sample applications in addition to a template. All of these 24 subjects used the traceability links between implementation steps in the templates and their corresponding instructions in sample applications' code, which were provided as searchable labels. For example, subject S_3 stated: *"Navigation to example applications is critical and a frequently performed step."* Interestingly, even S_7 and S_{23} mentioned in their debriefing interviews that they both used templates as the starting point to find the relevant code in the sample applications and, in this regard, templates were useful to them as well. Subjects also liked the syntactic resemblance between templates and Java code. For example, S_{16} stated: *"I like the format. In my opinion, being close to Java language is one of the best characteristics of the template format."* Further, S_8 said: *"I like that the code can be pasted directly into the Java program and be close to workable."*

Subjects also identified areas for improvement. Many suggested augmenting templates with 1) clickable traceability links instead of the searchable labels and 2) additional information from sample applications, API, or Javadoc. For example, S_{14} and S_{17} suggested including a brief description of each method and parameter in the template to avoid the context switch when delving into the sample code or Javadoc documentation. Further, several subjects noted that they mostly glanced over special

TABLE 4
Statistical Analysis of the Impact of Different Variables on the Subjects' Concept Implementation Times

Variable	Factor Level	Sample Size	Mean	Std Dev	Min	Med	Max	Kolmogorov-Smirnov (p-value)	Levene's test (p-value)	Cohen's d	t-test		
											Observed t	Critical t	p-value
Documentation Aid	Template	28	35.29	21.83	6.00	33.00	80.00	0.828	0.097	-0.46	-1.714	-1.674	0.046
	Documentation	28	46.82	28.13	10.00	42.50	99.00	0.706					
Concept Complexity	Simple	28	25.89	19.72	6.00	21.50	90.00	0.379	0.173	-1.46	-5.471	-1.674	<0.0001
	Complex	28	56.21	21.71	18.00	50.00	99.00	0.270					
Subject Experience	Experienced	28	33.54	23.09	6.00	31.50	90.00	0.817	0.184	-0.61	-2.278	-1.674	0.0133
	Moderate	28	48.57	26.2	10.00	42.00	99.00	0.717					

notations (e.g., "||" or REPEATED), finding them not useful. For example, subject S_{22} remarked that less choice would be preferred. On the other hand, subject S_{28} found that, whenever he saw "||", he wanted to consult Javadoc documentation to see what the alternative choices meant. Subject S_7 complained that the use of global variables in the templates made them too unstructured. Subject S_{17} would prefer starting from an executable example rather than a template that is not compilable. Subject S_{10} complained about false positives in the template which required him to study the sample applications. Some subjects remarked that they lacked prescribed strategies for using templates.

Subjects' comments on documentation. All 28 subjects confirmed that the provided documentation offered some useful information. Nineteen declared that the documentation contained enough information required for implementing the assigned concepts. The remaining nine said that information was missing and they had to extract it from the sample applications. Interestingly, 9 out of the 19 subjects for whom the documentation was sufficient still referred to the sample applications to gain additional confidence. For example, subject S_{27} stated: *"The documentation was enough. [...] I looked into the sample applications source code to become confident that my implementation is okay."*

Analysis of experiment questionnaires and debriefing interviews revealed that several subjects, e.g., S_8-S_9, S_{16}, S_{17}, S_{22}, mainly liked the presence of code snippets and examples in the documentation. Further, subjects S_8 and S_9 liked the correspondence between code snippets in the documentation for Content Assist and the provided sample applications, which allowed them to use the documentation as a mean to investigate the sample applications.

However, most subjects raised points against the provided documentation. In particular, subjects S_1, S_4, S_6, S_{16}, S_{19}, S_{22}, S_{26}, and S_{28} complained that they needed to filter out a lot of irrelevant material in the documentation in order to learn how to implement the assigned concepts. For example, subject S_{26} stated: *"Too much reading, I skipped most of the description to the example. I read the text only when certain things were not clear in the examples in the documentation."* Although most subjects liked the presence of code snippets in the documentation, subjects S_{11}, S_{13}-S_{14}, and S_{18} pointed out that the given documentation lacked a complete code example and they needed to find and assemble different code snippets to realize the assigned concept. Subjects S_4-S_6, S_8, S_{11}, S_{18}, and S_{23} mentioned that the documentation was missing some important information and they needed to investigate the sample applications to find it. Finally, subject S_{26} complained that the descriptions for some API elements were outdated in the documentation.

From these comments, we learned that subjects mainly like focused documentation that provides complete code examples for the desired concept instead of different code segments in different parts of the documentation and requiring the developer to find and assemble those parts to realize the concept.

We can summarize the likely reasons for why the T group outperformed the D group as follows: 1) The syntactic resemblance between templates and Java code was important for understanding and copying template code into the target application; 2) while the documentation was often incomplete, templates helped subjects to easily find the portions of sample application code and the API that were relevant to their task; and 3) with templates they avoided reading unnecessary text in the documentation.

TABLE 5
Impact of False Positives and False Negatives on the Concept Implementation Code

Concept	I	M	Impact
Context Menu	0	1	The false negative caused the implementation to fail.
Content Assist	13	1	The false positives caused an easy-to-fix compiler error and 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	1	The false negative caused the implementation to fail.

Template usage strategies. By analyzing the experiment questionnaires and debriefing interviews, we identified five main different strategies of how subjects used templates:

1. Subjects S_8, S_{12}-S_{13}, and S_{19} copied the code from the template into the target application without further investigating the sample applications. They mostly relied on compiler errors to detect errors.
2. Subjects S_3, S_5-S_6, S_{21}, and S_{26}-S_{28} copied the code from the template, then, after encountering some issues at compile time or runtime, they inspected the template code and sample applications to find out the reasons for these issues.
3. Subjects S_1, S_4, S_{11}, S_{15}-S_{16}, S_{18}, and S_{24} copied the code from the template into the target application, then they investigated the sample applications to refine that code.
4. Subjects S_2, S_7, S_{22}-S_{23}, and S_{25} used the template just as an entry to sample applications, then they copied the code snippets from the sample applications into the target application.
5. Subjects S_9-S_{10}, S_{14}, S_{17}, and S_{20} used the template just as an entry to sample applications, then they investigated the sample applications to learn their code; next, they wrote the code in the target application from scratch.

Thus, all except four subjects used templates together with sample applications. Interestingly, three out of these four subjects were in the experienced block. However, 18 subjects followed strategies 1-3, which means that they were able to directly use the code from templates. Ten subjects, who followed strategies 4-5, used templates only as a starting point to investigate sample applications. These observations confirm that templates can help users to focus only on the relevant parts of the sample applications code instead of investigating the whole applications.

Impact of false positives and false negatives. Table 5 indicates the numbers of false positives and false negatives and their impacts on each concept's implementation code. In general, false negatives prevented the full instantiation of a concept, as in the case of Context Menu and Table Viewer. False positives caused either compiler errors or runtime

errors (as in Content Assist) or polluted the concept instantiation with unnecessary code (as in Navigate). Interestingly, both of the subjects who had buggy implementations with templates, i.e., subjects S_8 and S_{19}, did not refer to sample applications during their tasks. In particular, the false positives in Navigate ended up in S_8's implementation.

Thus, our experiment reveals that subjects applying strategy 1 must be careful about false positives and false negatives. Further, it also shows that using templates together with sample applications helps detect code that is missing or unneeded.

7.5 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.

7.5.1 Internal Validity

The main threat to internal validity concerns the distribution of subjects over the concept implementation tasks. We addressed this threat by randomizing the assignment of subjects to tasks within the experience blocks. We also balanced the number of D and T treatments per block to avoid bias.

Because all subjects performed two concept implementations, this may have introduced learning effects from one task to the next. We minimized this threat by balancing the sequence of using the documentation aids and balancing the sequence of concept complexity. Another source of interference could be the subjects' prior knowledge and proficiency with the development environment. To reduce this threat, we provided them with basic tutorials and personalized training to bring each of them to the same level. We also banned the use of powerful features of Eclipse (e.g., debugger—which could have been used to understand sample applications). Further, none of the subjects had knowledge of FUDA templates prior to taking the tutorials. Finally, the subjects might have not followed our instructions; we reduced this risk by confirming in the debriefing interviews that they actually did follow the instructions provided in their task packages.

7.5.2 External Validity

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

The size and the complexity of concept instances used in this experiment is another source of threat to external validity. We selected concepts that could be implemented in a relatively short period of time (less than two hours); thus, the results are not representative of large, composite concepts. We assume that the task of implementing such concepts must be broken into smaller tasks of implementing smaller concepts handled by FUDA. Helping decompose large composite concepts into their components is future work.

Sample applications play a major role in the experiment. We assumed that template usage scenarios are likely to involve sample applications, especially to address false positives and negatives in the templates. To allow for a balanced comparison, we provided sample applications to both the T and D groups. Interestingly, the majority of subjects in both the T and D groups referred to the sample applications. The sample applications may not be representative; however, they were real applications obtained from open source using the same strategies developers use to obtain sample applications.

Another threat to external validity concerns the generalization from students to professionals. We minimized this threat by recruiting skilled Java programmers with at least three years of experience. Additionally, 21 subjects had one year or more of industry experience.

7.5.3 Construct Validity

To meaningfully compare templates and documentation, we should first make sure that all subjects used the prescribed aid (template or documentation) when creating their solutions. All subjects confirmed in their experiment questionnaires that this was the case.

Further, we need a clear definition of documentation. As described in Section 7.2, we used the standard framework documentation and made sure that the documentation was complete.

The use of sample applications in the experiment allowed us to obtain valuable qualitative data on template and documentation usage strategies. By using them, we were able to compare the use of templates with sample applications to the use of documentation with sample applications. We believe that the use of sample applications is more representative of real-world use of both documentation aids. Templates offer traceability links into the sample applications, which makes them more effective for exploring the sample applications. The documentation used did not have such links—a situation representative of real-world use of documentation.

The target applications could also influence the task completion time. In order to minimize this time, we kept these applications very small, less than 200 LOC, and asked subjects to familiarize themselves with them before performing the tasks. Further, the selection of the target applications should not introduce bias since both the T and D groups used the same target applications.

Finally, we need well-defined measures to compare the effectiveness of templates and documentation. We used the standard measures of implementation time and the functional correctness of the resulting code. Although the resulting implementations differed in qualities such as modularity and clarity, we did not take such differences into account to reduce subjectivity.

7.5.4 Replicability

We documented the methodology of this experiment, including the data collection and data analysis procedures. The complete experimental material, including the subjects' background questionnaires, experiment questionnaires, concept implementation templates, and the documentation used, is available online [15]. The target applications and the sample applications used in this study are open source and are also available online [15]. Thus, it should be possible to replicate the experiment.

8 Discussion

8.1 FUDA's Strengths and Weaknesses

The results of evaluating the template quality (cf. Section 6) indicate that FUDA can extract concept implementation templates with relatively high precision and recall from only two sample applications and execution scenarios. Furthermore, while collection of traces is manual, the processing of the traces is fully automatic. The instrumentation does not impose significant overhead on the application execution because only the API interactions rather than full traces are recorded. Given a set of applications and execution scenarios, the time needed to retrieve templates is mainly determined by the time needed to execute the scenarios on the applications. Thus, FUDA is particularly useful when the developer has some sample applications at hand and wants to see how they implement the desired concept. Furthermore, dynamic analysis detects the API elements that are actually being invoked. This characteristic of dynamic analysis is important because frameworks often use polymorphism and reflection, which can render the results of static analyses unusable. Finally, the second experiment (Section 7) revealed that using templates together with sample applications can reduce development time compared to when documentation is used. This result suggests that templates can be used instead of documentation, especially when no documentation is available.

Nevertheless, the approach has some drawbacks. Most importantly, it relies on the ability to find appropriate sample applications to generate templates. The template quality depends on the selection of the applications and concept invocation scenarios. In particular, creating the scenarios requires careful design to isolate the API instructions of interest in the context of multiple concepts. The approach is mainly applicable to GUI frameworks, where the desired concept can be identified visually and invoked interactively from the GUI. The approach may also be applicable to nongraphical frameworks, provided the concept of interest can be explicitly invoked from the sample applications' graphical or programmatic user interface; however, the effectiveness of the approach in these cases needs to be evaluated in future studies. Further, dynamic approaches require the setup of the runtime environment, which can be difficult in some situations. Finally, the results of dynamic approaches depend on input data. FUDA will retrieve the set of API instructions that are invoked in each execution, but may fail to retrieve optional API instructions. We discuss this issue in the following section.

8.2 Scenario Design Considerations

The nature of the concept of interest and the ways in which it is implemented by sample applications influence the results. Ideally, the concept's implementation is fixed, its invocation is easily delimitable by marking, and the sample applications have only this concept in common. In this case, FUDA will yield the best precision and recall results. In general, concepts contain variability (e.g., optional instructions), the invocation of a concept might not be easily demarcated, and the sample applications may have several concepts in common. For a variable concept, developers should select applications that differ in 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. In some cases, however, even marking cannot isolate the concept of interest. For example, for the concept GEF Figure, all figure drawing scenarios involved the palette and, hence, the extracted template contained some palette-related steps (cf. Section 6.4.2). Nonetheless, developers can deal with such issues by studying the actual sample application code that corresponds to template's implementation steps.

8.3 API Trace Slicing

The API trace slicing that is used in FUDA is significantly different from traditional dynamic program slicing techniques [23]. 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. 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 that the second call cleans up the object that was used in the first one. Similarly, a call that returns an object that is later used as a target or parameter in a subsequent call may be an invocation to a factory method. Clearly, identifying relevant events based on object relatedness may lead both to false positives and false negatives. For example, false positives may 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 quality evaluations, API slicing produced very good results in practice.

9 Related Work

Table 6 summarizes the related work discussed in this section and compares them with FUDA.

9.1 Framework Documentation Approaches

Cookbooks. Krasner and Pope [24] suggested organizing framework documentation as *cookbooks*. Each entry in the cookbook is a *recipe* that explains a typical usage scenario and provides stepwise guidance for its implementation. Pree et al. [25] introduced the concept of *active cookbooks* which include interactive elements to provide information on demand and to perform certain programming tasks semiautomatically. Cookbook recipes often focus on the extension points of a framework, also known as *hot-spots* (e.g., see the work by Froehlich et al. [66]).

Patterns. Johnson [26] proposed documenting frameworks using patterns. In his work, a pattern describes a situation that commonly occurs in the problem domain of the framework and discusses tradeoffs of possible solutions.

TABLE 6
Summary of Related Work Compared to FUDA

Related Work Category	Subcategories	Example Approaches	Main Aspects	FUDA
Framework Documentation Approaches	Cookbooks	Cookbooks [24], Active cookbooks [25]	Can provide rationale for design decisions, describe the API architecture, and verify conformance of client applications	Templates provide direct, specific guidance and can be generated automatically
	Patterns	Patterns [26], Design patterns [27], [28], Specialization patterns [29], Design fragments [30]		
	Specific Approaches	FCL [31], FSML [32]		
Supporting Framework Usage	Code Assistants	XSnippet [6], FrUiT [7], Prospector [33], PARSEWeb [34], Strathcona [35]	Provide fine grained guidance; easy to install and run because mostly static	Provides complete code snippet for a concept; more precision but also effort due to dynamic analysis
	API Comprehension Tools	Pattern Extractor [8], SpotWeb [9], [36], [37]	Present a general overview of the whole API	Provides a template targeting a specific concept
	Evolution Comprehension Tools	AURA [10], SemDiff [11], CatchUp! [38], Diff-CatchUp [39], [40], [41]	Describe general API changes over the framework evolution	Generates concept-specific templates for a single release
	Search-Based Tools	Assieme [42], XFinder [43], Exemplar [44]	Locate usage examples and additional information about a particular user-identified API element	No prior knowledge of source code required
Specification Mining	Static	MAPO [45], CHRONICLER [46], JADET [47], Permissive Interfaces [48], JIST [49]	Recover general API interaction protocols and potentially verify client application conformance	Recover the protocols used in the instantiations of a specific concept; call order is secondary
	Dynamic	[50], Perracotta [51], Javert [52], [53]		
Concept Location	Static	*Exploratory Approaches:* FEAT [54], JQuery [55], Active Models [56], Sextant [57]. *Lexical Code Searchers:* AspectBrowser [58], Find-Concept [59]. *IR-based Approaches:* SNIAFL [4], LSI-based [60]	Locate source code elements implementing a concept, using a query and/or trace intersection	Uses dynamic analysis and targets framework concepts (filters out application-specific content)
	Dynamic	Software Reconnaissance [61], DFT [62], SPR [63]		
	Hybrid	PROMESIR [3], SITIR [5], [64], [65]		

Subsequent work on *design patterns* [28] focused on creating and documenting framework designs.

Hakala et al. [29] proposed *specialization patterns* to define the specialization interface, i.e., the hot-spots, of a framework. Their approach is supported by a tool called *FRamework EDitor* (*Fred*). Framework developers can use Fred to define specialization patterns for a given framework, whereas application developers instantiate patterns for their applications. The tool provides a step-by-step task list derived from the specialization patterns and checks that the constraints of the framework are not violated.

Fairbanks et al. [30] introduced *design fragments*: implementation patterns encoding typical uses of a framework API to accomplish a given goal. Application developers can browse the catalog of available design fragments, select the ones that fit the application requirements and then bind the selected fragment to application code. A tool then checks if the application conforms to the fragment specification.

Specific approaches. Hou et al. [31] use the *Framework Constraint Language* (*FCL*) to specify all legal uses of a framework API. The semantics of FCL is based on first-order logic extended with set and sequence operations.

Antkiewicz et al. [32] propose *Framework Specific Modeling Languages* (*FSMLs*) to specify the concepts provided by a framework and how they relate to source code. The abstract syntax of an FSML decomposes a concept into a hierarchy of mandatory and optional *features*, represented as a *feature model* [67]. FSMLs can be used to describe how applications use the modeled concepts and to check for violations of framework rules.

Framework documentation approaches can be very effective in supporting application developers. In particular, documentation can provide insights on the rationale for design decisions and framework architecture, and specifications can support the automated verification of client applications. However, they require manual effort, are usually incomplete, and often become outdated as the framework evolves [2]. FUDA complements these approaches by providing specific guidance to concepts that may be missing from the documentation, provided sample applications are available. Further, templates can be automatically generated as the framework evolves.

9.2 Supporting Framework Usage

Several techniques have been proposed to address the problem of inadequate framework documentation. In particular, we discuss work that leverages framework and application source code to provide support for framework usage.

Code assistants. A number of approaches use existing framework applications to guide application developers during programming. Examples include *Prospector* [33], *PARSEWeb* [34], *XSnippet* [6], *Strathcona* [35], and *FrUiT* [7]. Given two API types τ_{in} and τ_{out} as a query, both Prospector and PARSEWeb mine for a sequence of calls that transform an object of type τ_{in} into another object of type τ_{out}. XSnippet and Strathcona are context-sensitive code assistants 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 their repositories and recommend relevant coding examples. FrUiT mines for frequent API usage patterns as *association rules* (e.g., $subclass\ A \Rightarrow call\ m$). Rules are then used to suggest relevant implementation steps.

Code assistants provide high quality, fine grained guidance in a given context. Additionally, their predominant use of static analyses allows them to cope with large

bodies of (often incomplete) framework and application code. FUDA templates are orthogonal in that they describe the whole implementation of a concept, which can span multiple methods and classes. Dynamic analysis also enables handling the highly polymorphic and reflective code of modern frameworks, but require the potentially cumbersome installation of sample applications.

Framework API comprehension tools. Several approaches extract API usage patterns from sample applications so that developers can understand how the framework API is to be used. *Pattern Extractor* [8] uses *formal concept analysis* [68] to extract Fred specialization patterns (cf. Section 9.1) from the source code of a framework and its sample applications. *SpotWeb* [9] gathers sample applications from open-source repositories and mines them to determine the *hot-spots* and *cold-spots* of a framework. Hot-spots are defined as frequently used API classes and methods, while cold-spots are those that are rarely used in sample applications. Schäfer et al. [36] exploit framework usage data to cluster elements of an API into building blocks that are typically used together. Finally, Bruch et al. [37] present an automated framework documentation approach that mines sample applications for *subclassing directives*, short descriptions of how to subclass a framework class or override its methods.

These approaches are similar to FUDA in their usage of sample applications to help developers understand an API. Nevertheless, while their goal is to present a general characterization of the whole framework API, FUDA targets the specific concept a user is interested in understanding.

Framework evolution comprehension tools. Framework APIs may evolve to address new requirements and to fix bugs [10]. Changes can break client applications and, thus, applications must evolve together with the framework. Several approaches have been proposed to automate this task, such as [38], [39], [11], [40], [41], [10].

CatchUp! [38] is a tool that records API refactorings applied when a developer evolves an API, and then replays the refactorings on the application code to be updated. *Diff-CatchUp* [39] uses a UML class model difference technique to recognize API changes. It also supports update by proposing plausible replacements for obsolete API elements based on framework examples. *SemDiff* [11] is a tool that analyzes how the framework itself was adapted to its changes and recommends similar adaptations for client applications. Schäfer et al. [40] infer framework usage change rules by observing how subsequent versions of framework applications and test cases use the framework. Kim et al. [41] use similarity in program element names to detect changes across versions. Finally, *AURA* [10] is a hybrid approach that combines call-dependency and text similarity analyses to infer change rules.

Framework evolution comprehension tools focus on describing and automating changes across versions. FUDA generates usage templates for a single version and does not help understanding how existing features changed. However, FUDA can help describe features that were newly introduced in a framework version, as long as sample applications are available.

Search-Based Tools. Another class of approaches is those supporting search of sample API usage in application code. For example, *Assieme* [42] is a special purpose web search engine with which users can search for specific API elements for a problem to get more information about them or to get sample usage. *XFinder* [43] is an extension of *Mismar*, a concept-oriented documentation toolset that focuses on code artifacts and their relationships. Given a Mismar concept implementation template, XFinder searches for instances of this template in its code base. *Exemplar* [44] is a tool that combines information retrieval and program analysis techniques. It uses a natural language query from the user and the API calls executed by an application to identify concepts.

Search-based approaches can locate additional information about an API element, but require that developers know about the element. FUDA extracts templates for a concept which can be located dynamically, without prior knowledge about the framework API.

9.3 Specification Mining

The term *specification mining* was first introduced by Ammons et al. [50] as an approach to discover protocols that clients should follow when interacting with an API. The large body of work on specification mining can be broadly classified into *static* and *dynamic* approaches.

Static specification mining. Static specification miners analyze the source code of sample applications and extract specifications of API protocols [45], [46], [48]. For example, *MAPO* [45] is a tool that searches open source repositories using a user-defined query characterizing an API by a method, class, or package, and applies data mining techniques to extract patterns of sequential method calls.

Some approaches, such as *CHRONICLER* [46] and *JADET* [47], not only determine protocols of method calls but also detect violations of those protocols in client applications. CHRONICLER infers *function precedence protocols* in the form of "a call to procedure q must always be preceded by a call to procedure p." Deviations from these protocols are reported as potential sources of bugs. Finally, other approaches like *Permissive Interfaces* [48] and *JIST* [49] accept an API type as input and generate a temporal specification as an automaton that encodes legal sequences of calls to that type.

Dynamic specification mining. Ammons et al. [50] pioneered dynamic specification mining. Their approach uses machine learning to mine temporal and data dependency specifications from dynamic traces of applications interacting with an API. *Perracotta* [51] and *Javert* [52] also mine for temporal specifications. Sankaranarayanan et al. [53] infer declarative specifications of the API behavior, such as raising exceptions, for user-defined concepts. A tool runs unit tests on the API and an inductive learner is used to obtain specifications expressed in Datalog/Prolog.

In contrast to specification mining, FUDA does not explicitly recover API protocol specifications, which are implicit in the templates. Specification mining is more important for libraries that require applications to follow protocols. Frameworks typically implement *inversion of control*, enforcing the protocol directly in framework code. Additionally, miners usually require a large number of runs to achieve good results. FUDA uses only two traces to make the technique attractive in practice.

9.4 Concept Location

Concept (or *feature*) *location* approaches aim to identify relevant portions of application source code with respect to some desired functionality or requirement. These approaches can be classified into *static*, *dynamic*, and *hybrid*.

Static concept location. These techniques extract data directly from application source code. We identify three main categories:

- *Exploratory approaches*, like *FEAT* [54], *JQuery* [55], *Active Models* [56], and *Sextant* [57], provide tools with which users can interactively explore or query application source code. The main assumption is that users have some knowledge about the concept implementation and the tools support exploration to build upon this initial understanding [4].
- *Lexical code searchers*, like *AspectBrowser* [58] and *Find-Concept* [59], allow users to lexically search application source code using regular expressions to locate concepts. For example, AspectBrowser searches programs using `grep`-like regular expressions and visually represents the results. The effectiveness of lexical code searches depends on good queries or strict naming conventions.
- *IR-based approaches*, like [4] and [60], apply information retrieval techniques to identify relevant parts of source code based on information available in identifiers and comments. For example, Marcus et al. [60] use natural language queries and *Latent Semantic Indexing* (LSI) to locate relevant source code elements. IR tools also rely on naming conventions, but are usually less sensitive to the quality of queries compared to lexical code searchers.

Dynamic concept location. These techniques analyze execution traces of use cases and map information back to source code. The major difficulty of dynamic approaches is that traces can be large and contain irrelevant events. Additional traces are often used to remove noise [5]. For example, in the *Software Reconnaissance* technique [61] users identify which traces contain and which traces *do not* contain the concept. Events in traces without the concept are used to filter out events in traces with the concept.

Dynamic Feature Traces (*DFT*) [62] follows a similar approach but additionally uses heuristics to rank program elements with respect to their relevance to a concept. *Scenario-based Probabilistic Ranking* (*SPR*) [63] targets concept location in large, multithreaded, object-oriented software systems. It uses processor emulation, knowledge-based filtering, and probabilistic event ranking.

Hybrid concept location. Hybrid approaches combine static and dynamic analyses and typically achieve better results than purely static or dynamic approaches [3]. Hybrid concept location was pioneered by Eisenbarth et al. [64]. In their work, traces of the invocation of concepts are used to generate a *concept lattice* [68] which is further refined using a manual static analysis based on the program dependency graph. Poshyvanyk et al. in *PROMESIR* [3] combine techniques presented earlier, the static LSI-based approach [60] and the dynamic approach based on probabilistic ranking of events SPR [63]. *SITIR* [5] tries to minimize the number of traces. It uses a single trace marked with the concept invocation and applies LSI to rank methods relative to a user-provided keyword query. Finally, Asadi et al. [65] locate concepts by splitting traces into cohesive segments representing concepts, using search-based optimization, LSI-based textual analysis of source code, and trace compression techniques.

The concept location techniques discussed focus on retrieving general concepts in application code rather than framework-provided concepts. Consequently, the results may contain application-specific content, which is irrelevant from the viewpoint of framework usage. FUDA removes application-specific information with event generalization, and focuses on API interaction traces instead of full application traces. API tracing also has the advantage of being more efficient since traces become much shorter. Additionally, the application of API trace slicing is unique to FUDA. In particular, although SITIR [5] uses trace marking to reduce the trace length, it does not apply slicing or any other techniques to identify relevant events that appear before or after the marked events.

10 Conclusions and Future Work

This paper presented FUDA, an approach for extracting templates from traces obtained by invoking concepts of interest in sample applications. FUDA was tested on 14 concepts with different characteristics, sourced from five widely used frameworks. Moreover, eight of the concepts corresponded to questions found on developer forums, representing real development issues. The experimental evaluation shows that, for the concepts considered, FUDA can extract templates with few false positives and negatives from only two sample applications per concept. Additionally, we reported on a user experiment with 28 programmers in which we compared templates with framework documentation in aiding the developers in performing concept-implementation tasks. The experiment revealed that the choice of templates versus documentation improved the implementation time. This result suggests that templates could be used instead of framework documentation, especially when no appropriate documentation is available. Templates can also be used by developers to narrow their focus to only relevant parts of the framework API or sample applications source code rather than investigating the whole framework API or sample applications to learn how to implement a concept of interest.

In the future, we plan to assess the quality of FUDA-generated templates for non-GUI frameworks and concepts. We also intend to compare the use of templates with sample applications against just sample applications. Finally, we want to explore combinations of static and dynamic analyses to extract templates with variants.

Acknowledgments

The authors would like to thank the anonymous reviewers for their thoughtful comments for improving this submission. They would also like to thank all 28 subjects who participated in the user study experiment.

REFERENCES

[1] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns, and Plug-Ins.* Addison-Wesley, 2003.

[2] M.P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE Software,* vol. 26, no. 6, pp. 26-34, Nov./Dec. 2009.

[3] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *IEEE Trans. Software Eng.,* vol. 33, no. 6, pp. 420-432, June 2007.

[4] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a Static Noninteractive Approach to Feature Location," *ACM Trans. Software Eng. and Methodology,* vol. 15, no. 2, pp. 195-226, 2006.

[5] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace," *Proc. IEEE/ACM 22nd Int'l Conf. Automated Software Eng.,* pp. 234-243, 2007.

[6] N. Sahavechaphan and K. Claypool, "XSnippet: Mining for Sample Code," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 413-430, 2006.

[7] M. Bruch, T. Schäfer, and M. Mezini, "FrUiT: IDE Support for Framework Understanding," *Proc. OOPSLA Workshop Eclipse Technology Exchange,* pp. 55-59, 2006.

[8] J. Viljamaa, "Reverse Engineering Framework Reuse Interfaces," *Proc. Int'l Symp. Foundations of Software Eng.,* pp. 217-226, 2003.

[9] S. Thummalapenta and T. Xie, "SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web," *Proc. IEEE/ACM 23rd Conf. Automated Software Eng.,* pp. 327-336, 2008.

[10] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "AURA: A Hybrid Approach to Identify Framework Evolution," *Proc. Int'l Conf. Software Eng.,* pp. 325-334, 2010.

[11] B. Dagenais and M.P. Robillard, "Recommending Adaptive Changes for Framework Evolution," *Proc. Int'l Conf. Software Eng.,* pp. 481-490, 2008.

[12] A. Heydarnoori, K. Czarnecki, and T.T. Bartolomei, "Supporting Framework Use via Automatically Extracted Concept-Implementation Templates," *Proc. European Conf. Object-Oriented Programming,* pp. 344-368, 2009.

[13] A. Heydarnoori, "Supporting Framework Use via Automatically Extracted Concept-Implementation Templates," PhD dissertation, Univ. of Waterloo, Canada, 2009.

[14] M.M. Salah, "An Environment for Comprehending the Behavior of Software Systems," PhD dissertation, Drexel Univ., 2005.

[15] Generative Software Development Lab, "FUDA Supporting Material," http://gsd.uwaterloo.ca/tse-fuda, 2011.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold, "An Overview of AspectJ," *Proc. European Conf. Object-Oriented Programming,* pp. 327-353, 2001.

[17] A. Villazón, W. Binder, and P. Moret, "Aspect Weaving in Standard Java Class Libraries," *Proc. Int'l Symp. Principles and Practice of Programming in Java,* pp. 159-167, 2008.

[18] "Eclipse Platform Community Forum," http://www.eclipse.org/forums/index.php?t=thread&frm_id=11, 2012.

[19] "GEF Community Forum," http://www.eclipse.org/forums/index.php?t=thread&frm_id=81, 2012.

[20] "Java 2D forum," http://forums.oracle.com/forums/forum.jspa?forumID=938, 2012.

[21] "Java Swing forum," http://forums.oracle.com/forums/forum.jspa?forumID=950, 2012.

[22] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting Empirical Methods for Software Engineering Research," *Proc. Guide to Advanced Empirical Software Eng.,* pp. 285-311, 2007.

[23] H. Agrawal and J.R. Horgan, "Dynamic Program Slicing," *Proc. Conf. Programming Language Design and Implementation,* pp. 246-256, 1990.

[24] G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80," *J. Object-Oriented Programming,* vol. 1, no. 3, pp. 26-49, 1988.

[25] W. Pree, G. Pomberger, A. Schappert, and P. Sommerlad, "Active Guidance of Framework Development," *Software—Concepts and Tools,* vol. 16, no. 3, pp. 136-145, 1995.

[26] R.E. Johnson, "Documenting Frameworks using Patterns," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 63-76, 1992.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[28] R.E. Johnson, "Patterns and Frameworks," *The Patterns Handbooks: Techniques, Strategies, and Applications,* pp. 375-382, Cambridge Univ. Press, 1998.

[29] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa, "Annotating Reusable Software Architectures with Specialization Patterns," *Proc. IEEE/IFIP Working Conf. Software Architecture,* pp. 171-180, 2001.

[30] G. Fairbanks, D. Garlan, and W. Scherlis, "Design Fragments Make Using Frameworks Easier," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 75-88, 2006.

[31] D. Hou, H.J. Hoover, and P. Rudnicki, "Specifying Framework Constraints with FCL," *Proc. IBM Centre for Advanced Studies Conf.,* pp. 96-110, 2004.

[32] M. Antkiewicz, K. Czarnecki, and M. Stephan, "Engineering of Framework-Specific Modeling Languages," *IEEE Trans. Software Eng.,* vol. 35, no. 6, pp. 795-824, Nov./Dec. 2009.

[33] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid Mining: Helping to Navigate the API Jungle," *Proc. Conf. Programming Language Design and Implementation,* pp. 48-61, 2005.

[34] S. Thummalapenta and T. Xie, "PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web," *Proc. Conf. Automated Software Eng.,* pp. 204-213, 2007.

[35] R. Holmes and G.C. Murphy, "Using Structural Context to Recommend Source Code Examples," *Proc. 27th Int'l Conf. Software Eng.,* pp. 117-125, 2005.

[36] T. Schäfer, I. Aracic, M. Merz, M. Mezini, and K. Ostermann, "Clustering for Generating Framework Top-Level Views," *Proc. 14th Working Conf. Reverse Eng.,* pp. 239-248, 2007.

[37] M. Bruch, M. Mezini, and M. Monperrus, "Mining Subclassing Directives to Improve Framework Reuse," *Proc. Working Conf. Mining Software Repositories,* pp. 141-150, 2010.

[38] J. Henkel and A. Diwan, "CatchUp!: Capturing and Replaying Refactorings to Support API Evolution," *Proc. Int'l Conf. Software Eng.,* pp. 274-283, 2005.

[39] Z. Xing and E. Stroulia, "API-Evolution Support with Diff-CatchUp," *IEEE Trans. Software Eng.,* vol. 33, no. 12, pp. 818-836, Dec. 2007.

[40] T. Schäfer, J. Jonas, and M. Mezini, "Mining Framework Usage Changes from Instantiation Code," *Proc. ACM/IEEE 30th Int'l Conf. Software Eng.,* pp. 471-480, 2008.

[41] M. Kim, D. Notkin, and D. Grossman, "Automatic Inference of Structural Changes for Matching across Program Versions," *Proc. 29th Int'l Conf. Software Eng.,* pp. 333-343, 2007.

[42] R. Hoffmann, J. Fogarty, and D.S. Weld, "Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers," *Proc. Int'l Symp. User Interface Software and Technology,* pp. 13-22, 2007.

[43] B. Dagenais and H. Ossher, "Automatically Locating Framework Extension Examples," *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 203-213, 2008.

[44] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A Search Engine for Finding Highly Relevant Applications," *Proc. ACM/IEEE 32nd Int'l Conf. Software Eng.,* pp. 475-484, 2010.

[45] T. Xie and J. Pei, "MAPO: Mining API Usages from Open Source Repositories," *Proc. ICSE Workshop Mining Software Repositories,* pp. 54-57, 2006.

[46] M.K. Ramanathan, A. Grama, and S. Jagannathan, "Path-Sensitive Inference of Function Precedence Protocols," *Proc. 29th Int'l Conf. Software Eng.,* pp. 240-250, 2007.

[47] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting Object Usage Anomalies," *Proc. Int'l Symp. Foundations of Software Eng.,* pp. 35-44, 2007.

[48] T.A. Henzinger, R. Jhala, and R. Majumdar, "Permissive Interfaces," *Proc. Int'l Symp. Foundations of Software Eng.,* pp. 31-40, 2005.

[49] R. Alur, P. Černý, P. Madhusudan, and W. Nam, "Synthesis of Interface Specifications for Java Classes," *Proc. Symp. Principles of Programming Languages,* pp. 98-109, 2005.

[50] G. Ammons, R. Bodík, and J.R. Larus, "Mining Specifications," *Proc. Symp. Principles of Programming Languages,* pp. 4-16, 2002.

[51] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining Temporal API Rules from Imperfect Traces," *Proc. Int'l Conf. Software Eng.,* pp. 282-291, 2006.

[52] M. Gabel and Z. Su, "Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces," *Proc. Int'l Symp. Foundations of Software Eng.*, pp. 339-349, 2008.
[53] S. Sankaranarayanan, F. Ivančić, and A. Gupta, "Mining Library Specifications Using Inductive Logic Programming," *Proc. Int'l Conf. Software Eng.*, pp. 131-140, 2008.
[54] M.P. Robillard and G.C. Murphy, "Representing Concerns in Source Code," *ACM Trans. Software Eng. and Methodology*, vol. 16, no. 1, pp. 3-38, 2007.
[55] D. Janzen and K.D. Volder, "Navigating and Querying Code Without Getting Lost," *Proc. Conf. Aspect-Oriented Software Development*, pp. 178-187, 2003.
[56] W. Coelho and G.C. Murphy, "Presenting Crosscutting Structure with Active Models," *Proc. Conf. Aspect-Oriented Software Development*, pp. 158-168, 2006.
[57] T. Schäfer, M. Eichberg, M. Haupt, and M. Mezini, "The SEXTANT Software Exploration Tool," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 753-768, Sept. 2006.
[58] W.G. Griswold, J.J. Yuan, and Y. Kato, "Exploiting the Map Metaphor in a Tool for Software Evolution," *Proc. Int'l Conf. Software Eng.*, pp. 265-274, 2001.
[59] D. Shepherd, Z.P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns," *Proc. Conf. Aspect-Oriented Software Development*, pp. 212-224, 2007.
[60] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," *Proc. 11th Working Conf. Reverse Eng.*, pp. 214-223, 2004.
[61] N. Wilde and M.C. Scully, "Software Reconnaissance: Mapping Program Features to Code," *J. Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49-62, 1995.
[62] A.D. Eisenberg and K.D. Volder, "Dynamic Feature Traces: Finding Features in Unfamiliar Code," *Proc. Int'l Conf. Software Maintenance*, pp. 337-346, 2005.
[63] G. Antoniol and Y.-G. Guéhéneuc, "Feature Identification: An Epidemiological Metaphor," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 627-641, Sept. 2006.
[64] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code," *IEEE Trans. Software Eng.*, vol. 29, no. 3, pp. 210-224, Mar. 2003.
[65] F. Asadi, M.D. Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A Heuristic-Based Approach to Identify Concepts in Execution Traces," *Proc. 14th European Conf. Software Maintenance and Reeng.*, pp. 31-40, 2010.
[66] G. Froehlich, H.J. Hoover, L. Liu, and P. Sorenson, "Hooking into Object-Oriented Application Frameworks," *Proc. Int'l Conf. Software Eng.*, pp. 491-501, 1997.
[67] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.
[68] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations.* Springer, 1999.

Abbas Heydarnoori received the PhD degree from the University of Waterloo, Canada, in 2009, and the MSc and BSc degrees from the Sharif University of Technology, Iran, in 2001 and 1999, respectively. After his PhD, he joined the Dynamic Analysis Group at the University of Lugano, Switzerland, as a postdoctoral researcher. He is currently an assistant professor in the Department of Computer Engineering at the Sharif University of Technology. His research interests include software reverse engineering and reengineering, dynamic program analysis, and component-based software development.

Krzysztof Czarnecki received the MSc degree in computer science at the California State University at Sacramento in 1994, and the Dipl-Inf and the PhD degree in computer science from Ilmenau University of Technology, Germany, in 1995 and 1999, respectively. He worked as a researcher at Daimler-Benz Research and Technology in Germany from 1995 to 2003. He is currently an associate professor in the Department of Electrical and Computer Engineering at the University of Waterloo and NSERC/Bank of Nova Scotia Industrial Research Chair in Requirements Engineering of Service-Oriented Software Systems. His research interests include domain-specific modeling, software product lines, and model synchronization.

Walter Binder received the MSc and PhD degrees and a venia docendi from the Vienna University of Technology, Austria. He is an associate professor on the Faculty of Informatics, University of Lugano, Switzerland. Before joining the University of Lugano, he was a postdoctoral researcher at the Artificial Intelligence Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland. His main research interests are in the areas of dynamic program analysis, virtual execution environments, aspect-oriented programming, resource management, and service-oriented computing.

Thiago Tonelli Bartolomei received the BEng degree from the State University of Campinas, Brazil (2003), and the MSc degree from the University of Applied Sciences Kiel, Germany (2006). He is currently working toward the PhD degree with the Generative Software Development Lab at the University of Waterloo, Canada. He held an IBM Centers for Advanced Studies PhD Fellowship with Toronto Labs from 2007-2009. His research interests include software reengineering and program analysis, with special focus on the role played by application programming interfaces in the software life cycle.