Supporting Framework Use via Automatically Extracted Concept-Implementation Templates

Abbas Heydarnoori, Krzysztof Czarnecki, and Thiago Tonelli Bartolomei

Generative Software Development Lab
University of Waterloo, Canada
http://gsd.uwaterloo.ca
{aheydarn, kczarnec, ttonelli}@gsd.uwaterloo.ca

Abstract. Application frameworks provide reusable concepts that are instantiated in application code through potentially complex implementation steps such as subclassing, implementing callbacks, and making calls. Existing applications contain valuable examples of such steps, except that locating them in the application code is often challenging. We propose the notion of *concept implementation templates*, which summarize the necessary implementation steps, and an approach to automatic extraction of such templates from traces of sample applications. We demonstrate the feasibility of the template extraction with high precision and recall through an empirical study with twelve realistic concepts from four widely-used frameworks. Finally, we report on a user experiment with twelve subjects in which the choice of templates vs. documentation had much less impact on development time than the concept complexity.

1 Introduction

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

Many existing frameworks are difficult to use because of their large and complex *Application Programming Interfaces* (APIs) and often incomplete user documentation. To cope with this problem, application developers frequently apply the *Monkey See/Monkey Do* rule [1]: "Use existing framework applications as a guide to develop new applications". Understanding how an existing application implements a concept requires the ability to quickly locate those parts of the application code that implement the concept. Unfortunately, locating the concept implementation can be challenging since the implementation is often scattered in the application code and tangled with the implementation of other concepts.

Several tools have been proposed to address this challenge. *Framework usage comprehension* tools such as Strathcona [2] and FrUiT [3] apply static analysis to the source code of sample applications and allow retrieving code snippets or usage rules for a particular API element. These tools can be very helpful to understand concept implementations, but require the developer to know at least the names of some of the API elements involved. They are less helpful if the developer has only a high-level idea of the concept that needs to be implemented or if the concept spans multiple classes or both. *Concept location* tools such as SNIAFL [4] or SITIR [5] can be used to locate the code implementing a concept of interest identified by higher-level characteristics such as usage scenarios or domain terms. These tools do not focus on framework API usage, however: the code identified will still include many application-specific elements that are irrelevant from the viewpoint of framework usage.

To address the above issues, we propose the notion of *concept implementation templates* and *FUDA* (*Framework Understanding through Dynamic Analysis*), an approach to automatic extraction of such templates from traces of sample applications. A concept implementation template is a tutorial-like code example summarizing the *implementation steps* necessary to instantiate a given concept. Such a template can be used as a starting point to further investigate the concrete concept implementations in the sample applications.

The FUDA template extraction approach works by invoking the concept of interest in at least two different contexts in one or more sample applications, and recording all runtime interactions between the application code and the framework API. For instance, given the context menu in an Eclipse view as the desired concept, each trace could be collected by invoking a context menu in a different Eclipse view. The collected traces are then intersected. The calls in the intersection provide the basis for generating a template that specifies which packages to import, framework classes to subclass, interfaces to implement, and operations to call in order to implement the concept.

We have implemented the extraction approach as a tool for Java and used this tool in a study to evaluate the quality of the extraction. The study shows that the approach can produce templates with relatively few false positives and false negatives for realistic concepts by using only two sample applications.

Furthermore, we conducted a user experiment with twelve subjects comparing templates to framework documentation. For the studied sample, no statistically significant difference between using templates and documentation in terms of implementation time and number of introduced bugs could be detected. 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.

The contributions of the paper include (1) the notion of automatically extracted concept implementation templates, (2) an approach to automatic extraction of such templates from sample applications, (3) a prototype implementation of the extraction approach, (4) a study evaluating the precision and recall of the extraction approach, and (5) an experiment evaluating the usefulness of templates in comparison to framework documentation in assisting application developers.

In the remainder, we introduce our running example (Section 2) and present the notion of templates (Section 3). We then describe the extraction approach (Section 4)

and present its evaluation (Section 5). Finally, we discuss several aspects of FUDA (Section 6), compare it with related work (Section 7), and conclude (Section 8).

2 Running Example

As an example, consider the code implementing a context menu using JFace (Figure 1). The menu is located in `SampleView`, which is a visual component that displays trees using a `TreeViewer` (l. 36). The code was generated using one of Eclipse's wizards. The lines implementing the context menu are marked by •. The lines marked by ◦ implement a Welcome 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 illustrates some of the challenges in locating concepts in code. The implementation of the menu is tangled with the implementation of the view and it involves a complex interaction of several objects, namely view, menu manager, menu listener, menu, actions, and separators. To complicate 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 GUI of a sample application may be easy, locating its implementation in the application code is often challenging and time consuming.

3 Concept Implementation Templates

A template for our context menu example is shown in Figure 2. The template was generated from two traces collected by invoking the context menu in two sample applications: `SampleView` (Figure 1) and `Console`, which is part of Eclipse. The generated template has the form of a tutorial-like example in Java-based pseudocode.

Templates specify the following implementation steps: packages to import (l. 1–8 in Figure 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). Note that the specified steps involve only the elements of the framework API. For example, the method calls `makeActions()` and `hookContextMenu()` in Figure 1 are specific to that particular implementation and are not reflected in the template. The involved elements may be entirely framework-defined, e.g., the implementation of `Separator`, which is instantiated in line 11, resides in framework code. Alternatively, the elements may also reside in the application code, provided that they are *framework-stipulated*. 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 framework-stipulated classes. For example,

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

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

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

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

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

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

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

`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 (i) call nesting, e.g., `add()` is called directly or indirectly by `menuAboutToShow()` (l. 12); (ii) call order, e.g., the menu listener is added to the menu manager (l. 27) before creating the menu (l. 28); and (iii) 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 Figure 2) is due to multiple calls to `add()` with different arguments (l. 213 and 215 in Figure 1). Second, what appears to be a local variable declaration in Java, such as `appAction` (l. 21), actually has global meaning in the template. For that reason, `appAction` can be used as a method argument in another method scope (l. 12).

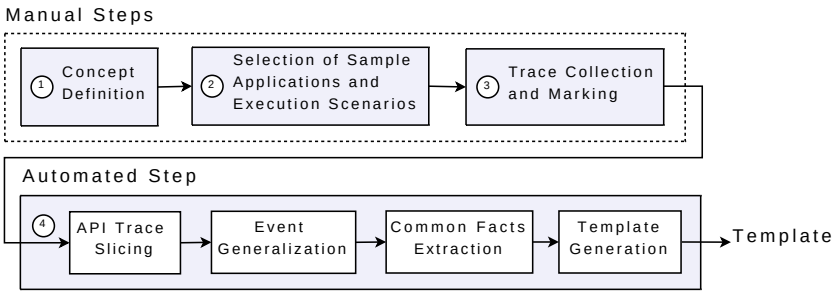

Fig. 3. FUDA overview

A template extracted by FUDA is an approximation of the necessary implementation steps, and it can be incomplete or unsound or both. In particular, implementation steps can be missing (*false negatives*) or unrelated steps (*false positives*) can be present in some cases. 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, such a step may get filtered out. Furthermore, FUDA relies on the assumption that input traces show the execution of the concept of interest in different contexts. For example, `SampleView`, which uses `TreeViewer`, and `Console` provide entirely different contexts for the context menu. In contrast, a template generated from `SampleView` and a view containing a `TableViewer`, which is graphically different from the `TreeViewer` of the `SampleView`, would also contain calls to `setContentProvider` and `setLabelProvider`. Although the calls to these two methods are not related to the context menu, they are contained both in code instantiating `TreeViewer` and in code instantiating `TableViewer`. 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. The traceability links between the implementation steps in the template and the corresponding steps in the application code, as shown in Figure 2, can support this task.

4 The FUDA Template Extraction Approach

From a user perspective, FUDA's process consists of the four steps depicted in Figure 3. The first three steps are performed manually by the user; the last one is a composition of several automated sub-steps. We describe each step in detail next.

4.1 Concept Definition

A framework-provided concept may be implemented by one or more objects. FUDA can produce implementation templates covering the entire life cycle of a concept, which

involves concept creation (creating and setting up its implementation objects), concept invocations (calling the objects) and concept destruction (tearing down and disposing the objects). The following *concept-defining question* asks for the entire life cycle of a concept: "*How does one implement a context menu in an Eclipse view?*" Alternatively, FUDA can also produce implementation templates covering individual concept invocations, as exemplified by this question: "*What events occur when a user clicks on a figure?*"

4.2 Selection of Sample Application(s) and Execution Scenarios

FUDA requires one or two applications implementing the concept of interest. It also needs two execution scenarios, each invoking an instance of the concept. Graphical concepts can be directly invoked, but FUDA is also applicable to non-graphical frameworks, as long as the concept of interest can be explicitly invoked from the sample applications' graphical or programmatic user interface.

The applications and the scenarios should be selected to achieve one or more of the following goals: (1) The scenarios are concept-focused: ideally the majority of the executed instructions are part of the concept. (2) The concept is invoked separately from others as part of the scenario and the invocation can be explicitly marked. (3) Each concept instance is invoked in a different context. A single application may already support the third goal, e.g., an application implementing a context menu in two different views would suffice. Because FUDA works by intersecting traces of the different executions, the more the contexts differ, the lower the possibility of false positives. For the same reason, it is important to select scenarios that contain a similar variant of the concept, which minimizes false negatives. For example, if a variant of the context menu concept with a separator is desired, scenarios that contain separators should be selected.

4.3 Trace Collection and Marking

This step involves running each sample application under a *tracer* and invoking the concept of interest. The user specifies 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 logs all calls that occur at the boundary between the application and the framework, which results in a *framework API interaction trace*, called *API trace* for short. If possible, pinpointing the moments before and after the concept invocation will improve the template extraction results, which is in fact essential for concepts whose defining question deals with the response to an event. For the context menu example, pinpointing amounts to instructing the tracer to *mark* subsequent events right before opening the menu and instructing it to stop marking right after the menu is open. If the moments before and after concept invocation cannot be pinpointed, the whole trace is marked. Concepts invoked through a programmatic interface can use the tracer tool to indicate the begin and end of the concept execution.

The 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. This boundary consists of (1) all calls to application methods and constructors that are framework-stipulated and (2) calls to API-defined methods

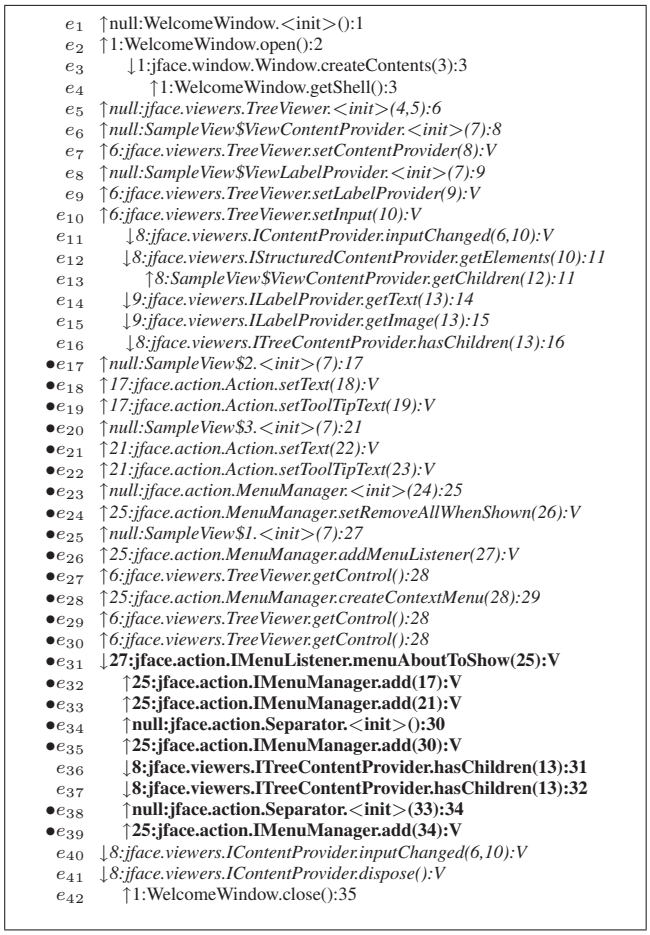

Fig. 4. Framework API interaction trace

and constructors from within the application. Each event has one of two directions: (1) an event is *outgoing* iff the call is made from within the application code; and (2) an event is *incoming* iff the call is made from within the framework code, i.e., a callback.

The complete API trace produced by running `SampleView` from Figure 1 and invoking its context menu is shown in Figure 4. Events are denoted as *D O:n(P):R*, where *D* represents the direction of the event, with "↓" for incoming and "↑" 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. 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.

Most of the events in Figure 4 can easily be traced back to their corresponding code lines in Figure 1. The events e_1–e_{30} are generated when the `createPartControl()` is called, e.g., e_1 is due to line 191. The actual call to `createPartControl()` is not traced because it resides in `eclipse.ui`, which is not part of JFace. The calls in l. 209 and l. 210 are not traced for the same reason. Indentation denotes *event nesting*. For example, events e_3 and e_4 were generated in the control flow of the call to `welcomeWindow.open()` (l. 192). Anonymous classes are denoted by numbers separated from their host classes by `$`, e.g., e_{17} constructs `action1` (l. 220).

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

4.4 Automated Trace Processing

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

API Trace Slicing. The marked trace region (bold face) in Figure 4 contains the calls that occurred when the context menu was being opened. Selective marking improves the results by delimiting the interaction between application and framework that is relevant for the concept of interest. If the goal is to understand the complete life cycle of the concept, however, it is necessary also to consider calls related to the initialization and clean-up of the involved objects, which are not reflected in this marked region. For example, e_{17}–e_{22} create and initialize the context menu's actions.

We use *API trace slicing* to identify these additional calls based on the marked region. API trace slicing identifies all the unmarked calls in the input trace that involve any of the objects that are also involved in a marked call. The precise definition is based on the *object-connectedness* of two events. The two events e_i=D_i O_i:$n_i(P_i)$:R_i and e_j=D_j O_j:$n_j(P_j)$:R_j are object-connected iff they share any target, parameter, or returned objects, i.e., $(\{O_i, R_i\} \cup P_i) \cap (\{O_j, R_j\} \cup P_j) \setminus \{\text{null}, \text{V}\} \neq \emptyset$. Also, let *object-relatedness* be the transitive closure of object-connectedness. Then, a trace slice is defined 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 Figure 4, the unmarked events that are object-related to the marked ones are typeset in italic font. 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. Note that slicing eliminates the steps implementing the Welcome window (e_2–e_4, e_{42}), which are unrelated to the context menu. As can be seen in Figure 4, API trace slicing is an approximation of the actual dependencies between API calls. The approximation worked perfectly for the real framework APIs in our evaluation (Section 5), however: there was not a single false negative due to slicing. Slicing is optional since some concepts focus on the invocation only, in which case no slicing is needed and only the marked events are further processed. Also, if marking is not used, FUDA will process the entire trace.

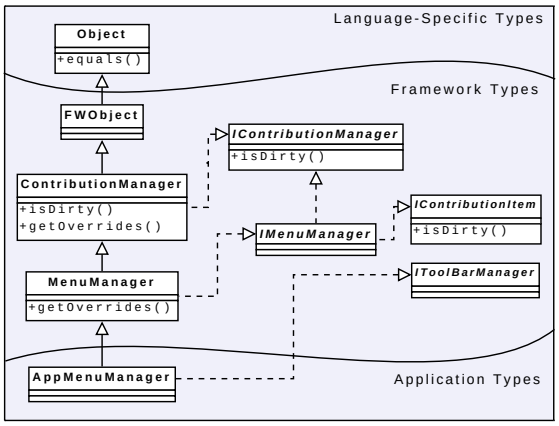

Fig. 5. Boundaries of application, framework, and language-specific types

Event Generalization. The generalization procedure allows the next processing stage to compare traces in terms of framework API types. This procedure is a static analysis that replaces the application-specific names of events with appropriate framework names. For example, the fully qualified name of e_6 in Figure 4 (`SampleView$-ViewContentProvider.<init>`) is application-specific and generalization replaces it by `[jface.viewers.IStructuredContentProvider, jface.-viewers.ITreeContentProvider].<init>`. The two names in brackets refer to the framework interfaces that `ViewContentProvider` implements (l. 99). Event generalization treats calls to instance methods, constructors, and static methods differently. We will explain it using Figure 5, which shows some menu-related classes.

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 topmost types since 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. For example, for a call to the constructor of `AppMenuManager`, the procedure identifies `MenuManager` and `IToolBarManager` as the generalized framework types. The rationale for minimum generalization for constructors is that selecting the topmost types, even if only 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 Figure 5,

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.

Common Facts Extraction. Three types of facts are extracted from each generalized trace: *event occurrence facts*, *event nesting facts*, and *event dependency facts*. The first represents the occurrence of interaction events in the generalized trace, while the remaining represent the existence of certain relationships among events. Then, *common facts* are computed as intersections of the extracted fact sets across the generalized traces. Figure 6 presents different kinds of common facts extracted from two generalized traces for `SampleView` and `Console` example applications.

Event occurrence facts, called *event facts* for short, are the names of the methods and constructors that were called at the application-framework boundary and the corresponding call directions (Figure 6(a)). They abstract away the numbers of occurrences, object IDs, and parameter and return types of the corresponding calls. The rationale is that two methods with the same name but different parameter or return types or numbers of parameters are likely to be conceptually equivalent within an API. An event fact D $t.n$, where t is a type name, is extracted from a generalized trace iff the trace contains one or more events D O_i:[...,t,...].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, a_2 is extracted from the generalized trace due to its events corresponding to e_{18} or e_{21} in Figure 4. The events in Figure 4 that match the common event facts in Figure 6(a) are marked by •. The remaining events are effectively filtered out as they were unique to this trace.

Event nesting facts record the calling context for outgoing calls (Figure 6(b)). A nesting fact $a_i \rightarrow a_j$, where a_i and a_j are event facts, is produced whenever the generalized trace contains two events e_k and e_l such that (i) e_k and e_l match a_i and a_j, respectively; (ii) e_l is outgoing; and (iii) e_l is directly nested in e_k in the trace.

Event dependency facts represent call sequence and object passing patterns. There are nine dependency fact types: 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 TT(a_i, a_j), where a_i and a_j are event facts, is produced whenever the generalized trace contains two events e_k and e_l such that (i) e_k and e_l match a_i and a_j, respectively; (ii) e_k precedes e_l in the trace; and (iii) both e_k and e_l have the same object as target. The analogous definitions for the remaining dependency fact types are obtained by modifying the third condition. For example, if the return object ID of e_k is used as a parameter in e_l, the resulting fact type is RP(a_i, a_j). Dependency facts indicate sharing of objects and object passing; e.g., PR and TR may represent the registration of an object with a framework and subsequent retrieval. After the common facts are computed, the event facts that originated from the same generic events (because of multiple type names due to generalization) are collapsed and the affected common nesting and dependency facts are updated accordingly.

Template Generation. This section sketches the main steps of the template generation algorithm. Interested readers can refer to [6] for further details. The inputs to the al-

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

(a) Common event occurrence facts

$a_{10} \rightarrow a_{11}$
$a_{10} \rightarrow a_{12}$

(b) Common nesting facts

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

(c) Common dependency facts

Fig. 6. Common facts

gorithm are the three sets of common facts and the generalized traces. The common facts determine the overall structure of the template, and the traces are used to extract additional details as needed. The algorithm executes the following steps.

Create Classes. A class is created for each group of incoming event facts that are related by TT dependencies. The corresponding constructor calls are assigned using RT dependencies. For example, the class in l. 9 (Figure 2) is created for the fact a_{10}, which does not participate in any TT dependencies and thus forms its own group. The corresponding constructor call is a_6, due to RT(a_6,a_{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. 15 is created for a_1, a call to the constructor of the abstract class `Action`.

Create Methods and Constructors. For each incoming event fact 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 a_{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 methods based on the common nesting facts. For example, the nesting fact $a_{10} \rightarrow a_{12}$ places the call in l. 12. The generalized traces are then consulted to see whether the calls are repeated in a given calling context. For instance, the call to a_{12} is marked as `REPEAT` since a_{12} was called multiple times in every trace within the control flow of `menuAboutToShow()`. The additional

class `SomeClass` (l. 17) is created to host calls corresponding to outgoing event facts for which no calling contexts are specified by the nesting facts. Within each method, calls are sorted in an order determined by the dependency facts. For example, the call order in the context menu template was obtained as a topological sort on the graph with event facts as nodes and the dependency facts as directed edges. Since 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.

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 superclass and the interfaces for that class.

Generate Class and Variable Names. Each class is named by prepending `App` to its superclass name or one of its interface names if no superclass is present. 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.

Broadcast Variables. The dependency facts, except RR, are used to identify object passing between calls. For example, `appAction` is passed as a parameter to `add` in l. 12 because of RP(a_1,a_{12}). New variables are introduced as needed. 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. Package imports are identified based on the fully qualified type names of the event facts.

5 Evaluation

The approach was evaluated through an experiment to assess the template extraction process (Section 5.1) and an additional experiment to analyze the usage of templates in the implementation of framework-provided concepts (Section 5.2).

5.1 Evaluation of Template Extraction

Experimental Design. This evaluation was designed to verify that *FUDA can extract templates with high precision and recall from only two traces and two sample applications*. We aim at keeping the number of traces per concept as small as possible since the collection of traces represents manual effort. In particular, the installation and execution of sample applications can be cumbersome. Also, in an earlier experiment [7] we showed that additional applications cause templates to concentrate on the minimal common implementation steps, without much improvement in terms of false positives or negatives. The evaluation consists of the following steps.

Selection of Frameworks and Concepts. The evaluation includes four widely used frameworks (Table 1). The JFace and Eclipse concepts, except Focus, were selected as representative for FUDA based on the authors' prior familiarity with these frameworks.

Table 1. Experimental data (* indicates concepts from developer forums)

Framework	Concept	Defining Question	Properties					Sample Applications		Precision and Recall Results									
										No Slicing					With Slicing				
			In scope	Slicing	Frequent	Simple	Composite	Name	Source	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	X	Tree View	Eclipse Wizard	15	0 (4)	1 (1)	100 (73)	94 (92)	15	0 (4)	1 (1)	**100 (73)**	**94 (92)**
								Console	Eclipse UI										
	Toolbar Button	How to add a button to a view's toolbar?	X	X	X	X	X	Pkg Explorer	Eclipse JDT	18	5 (14)	3 (3)	72 (22)	81 (57)	13	1 (9)	3 (3)	**92 (31)**	**80 (57)**
								Crosscutting Comparison	AJDT										
	Content Assist	How to develop a content assistant in a text editor?	X	X	–	–	X	Java editor	Eclipse JDT	46	27 (30)	1 (1)	41 (35)	95 (94)	32	13 (16)	1 (1)	**59 (50)**	**95 (94)**
								JSP editor	Eclipse WTP										
Eclipse	Table Viewer	How to develop a table viewer?	X	–	X	–	X	Editor List	eclipse-plugins	39	0 (23)	0 (0)	**100 (41)**	**100 (100)**	–	–	–	–	–
								Table View	Eclipse Wizard										
	Tree Viewer	How to develop a tree viewer?	X	–	X	–	X	LDAP Brwsr	eclipse-plugins	45	0 (29)	1 (1)	**100 (36)**	**98 (94)**	–	–	–	–	–
								Tree View	Eclipse wizard										
	Navigate	How to create the tree navigation buttons in a view's toolbar?	X	X	–	X	X	KTreeMap	SourceForge	40	10 (20)	0 (0)	75 (50)	100 (100)	38	8 (18)	0 (0)	**79 (53)**	**100 (100)**
								SVN Repository	Subclipse										
	Focus*	What events happen by clicking on a view's titlebar?	X	–	X	X	–	LDAP Brwsr	eclipse-plugins	4	0 (0)	0 (0)	**100 (100)**	**100 (100)**	–	–	–	–	–
								Editor List	eclipse-plugins										
GEF	Select*	What events happen by clicking on a figure?	X	–	X	X	–	Flow	GEF Examples	7	0 (3)	0 (0)	**100 (57)**	**100 (100)**	–	–	–	–	–
								Shapes	GEF Examples										
	Figure*	How to draw a figure in a GEF editor?	X	X	X	–	X	Flow	GEF Examples	83	25 (75)	0 (0)	70 (10)	100 (100)	68	10 (60)	0 (0)	**85 (12)**	**100 (100)**
								Shapes	GEF Examples										
	Connec-tion*	How to draw a connection between two figures?	X	X	X	–	X	Flow	GEF Examples	91	26 (82)	0 (0)	71 (10)	100 (100)	76	10 (66)	0 (0)	**87 (13)**	**100 (100)**
								Shapes	GEF Examples										
	Title-Bar Color*	How to change the color of a GEF editor's title-bar?	–	–	–	X	–	–	–	–	–	–	–	–	–	–	–	–	–
								–	–										
Java 2D	Moving Shapes*	How to draw shapes and let the user drag them?	X	X	X	X	X	GTEditor	Google Code	25	7 (16)	4 (4)	72 (36)	82 (69)	18	3 (9)	4 (4)	**83 (50)**	**79 (69)**
								GeoSoft	Google Search										
	Circle Drawing*	How to draw a red circle on a black background?	X	X	X	X	X	JHotDraw	SourceForge	12	4 (9)	0 (0)	67 (25)	100 (100)	10	2 (7)	0 (0)	**80 (30)**	**100 (100)**
								Scribble	Google Search										
	Rounded Image*	How to make the corners of an image rounded?	–	–	X	X	–	–	–	–	–	–	–	–	–	–	–	–	–
								–	–										

G: Template size; I: Num of false positives; M: Num of false negatives; P: Precision; R: Recall.

The remaining concepts (i.e., their defining questions) were sampled from developer forums of the respective frameworks and FUDA steps were performed for them without much prior knowledge of the corresponding frameworks.

The concepts were selected to cover a variety of characteristics, namely being *in scope* of FUDA's intended usage or not, requiring *slicing* or not, being *frequent* among existing applications or *rare*, being *simple* or *complex* in terms of implementation complexity measured as template size, or being *composite* in the sense of consisting of several variable subsets of implementation steps or *atomic*, if only a fixed set of steps is involved.

Tree Viewer and Table Viewer are the concepts in Table 1 where slicing was not used since the scenario involving view opening and closing spanned the entire view life cycle. Thanks to trace slicing, FUDA also works well for concepts having life cycles spanning beyond the marked trace region, which are those shown with "X" in the Slicing column. We also included two concepts, Focus and Select, for which only the marked region is used since the defining question asked for the response to an event. Obviously, FUDA works best for frequent concepts, in which case finding sample applications is likely easy. It may be applicable to rare concepts, too, if the user has already identified one or two applications with appropriate execution scenarios. For example, users may apply FUDA to find out the implementation of a rare concept that caught their eye in an existing application. Also, concepts that may appear rare at first might not be rare after all. For example, the choice of red and black in Circle Drawing may be rare, but setting background and figure colors is not. Most of the considered concepts are composite as they include variable parts. For example, a context menu may or may not include a separator. Concepts with only few implementation steps tend to be atomic and more complex ones are usually composite. Finally, Rounded Image and Title-Bar Color are out of scope because it is very unlikely to find applications and scenarios satisfying the three goals from Section 4.2 for them.

Selection of Applications and Execution Scenarios. The sample applications come from different sources (Table 1), such as framework-packaged examples for Eclipse's Graphical Editing Framework (GEF), applications listed in online repositories, e.g., `eclipse-plugins.org`, or part of a larger familiar environment, e.g., Java Development Tools (JDT). Application selection involved (i) reliance on prior familiarity with a given application (mostly for JFace and Eclipse), (ii) browsing and running the standard examples (e.g., GEF), (iii) searching or browsing in application repositories (e.g., GTEditor for Moving Shapes was identified on Google Code by the search keyword "shape" and seeing a screenshot of a drawing editor), or (iv) tips by others (e.g., WTP was suggested by a colleague). Selecting the applications for each concept took anywhere from no time for Eclipse JDT or wizards thanks to prior familiarity to up to an hour of searching and browsing for `eclipse-plugins.org` or `SourceForge.net`. The selection process had a significant learning effect: familiarity with framework-packaged examples or applications inspected for a given concept significantly reduced the selection time for the next concept. Some execution scenarios were already specified by the defining questions, e.g., "*How does one draw a figure in a GEF editor?*" In other cases, an action invoking the concept of interest had to be identified, e.g., the opening action for Context Menu.

Trace Collection. The tracer used to collect the traces was implemented using AspectJ. The user had to specify the packages in which the framework of interest and the sample application reside. All concepts in Table 1 involved trace marking, except Tree Viewer and Table Viewer for which full traces were used. Note that only the calls at the application-framework boundary are traced, which are drastically fewer than all the calls involved in the implementation of a concept. As a result, API tracing is quite efficient. For example, tracing all of GEF (`org.eclipse.gef.*`) was almost unnoticeable when using GEF applications on a laptop with a single-core Pentium M 1.6MHz processor, 1GB of RAM, and Windows XP. The applications ran two to three times slower when all of Eclipse was traced (`org.eclipse.*`), however. Collecting a single trace took anywhere from several seconds to a few minutes.

Template Generation. The template extraction algorithms described in this paper were implemented as an Eclipse plug-in. This tool was then used to generate the templates.

Development of Reference Templates. The precision and recall of the generated templates were calculated against reference templates. For each concept in scope a *mandatory* and an *optional* reference template were created. Mandatory reference templates represent the set of mandatory implementation steps, i.e., the ones that are essential to the instantiation of a concept: if the step is removed, the concept does not work as expected. For example, without calling the method `createContextMenu()` (l. 28 in Figure 2) a context menu cannot be realized. For concepts that relate to the response to an event, such as Focus and Select, the mandatory steps are the ones that always occur as a result to the event. Optional reference templates additionally include steps that are not essential but that are relevant to the concept and were present in the sample applications. For instance, Context Menu's optional reference template includes calls to create and register separators.

Reference templates were carefully created to minimize threats to the validity of the results. For all concepts, reference templates were created using documentation found online (usually third-party articles or solutions posted in forums or both) and manually inspecting sample applications. In order to guarantee their correctness, reference templates were used in the creation of sample implementations. The determination of mandatory steps was mostly obvious; dubious cases were verified by removing the step from the sample implementation and testing. Reference templates were then compared against the generated ones to identify optional features present in the sample applications. Each non-mandatory step found in the generated template was examined and classified as optional, if it was relevant to the concept, or *irrelevant*, otherwise. If not clear, we were conservative and the step was considered irrelevant.

Calculation of Precision and Recall. The calculation of precision and recall is based on counting the basic implementation steps contained in a template: subclass declarations, interface implementation declarations, method implementations (except `someMethod()`), method calls, and constructor calls. These steps are the main elements of a template. Call sequence and parameter passing patterns are considered supplementary information that makes templates more readable. The calculation of precision and recall is based on determining three numbers: G is the number of all implementation steps in the generated template; (ii) M is the number of steps present in the reference template

but missing in the generated template (i.e., false negatives); and (iii) I is the number of steps incorrectly present in the generated template, but absent in the reference template (i.e., false positives). Precision is calculated as $P = (G-I)/G$, and *recall* is calculated as $R = (G-I)/(G-I+M)$.

Experimental Results. The precision and recall results are given in Table 1. For the concepts with slicing, we also include the numbers obtained by using full traces without slicing, for comparison. The final numbers are marked in bold, with precision ranging between 59% and 100% and recall ranging between 79% and 100% when optional reference templates are used. When mandatory reference templates are used (in parentheses), precision ranges between 12% and 100%, and recall ranges between 57% and 100%. Note that users are likely interested in templates similar to optional reference templates; the mandatory reference templates are used to establish a lower bound on the precision and recall. In the following, we concentrate on the results for optional reference templates.

In general, false positives were more frequent than false negatives. False positives were due to similarities among the sample applications that extend beyond the concept of interest. For example, the single false positive for Toolbar Button was due to calls to the method `IShellProvider.getShell()`, which are frequently used in Eclipse views. Slicing improved precision by eliminating between 20% and 80% of false positives, except for Context Menu, for which the sample applications were different enough to achieve 100% precision without slicing. Slicing was particularly useful for Figure and Connection since all GEF editors use common parts such as palette and action bar. While steps related to action bar were eliminated by slicing, some palette-related steps remained since palette was involved in all figure drawing scenarios.

The false negatives for JFace concepts represent the case when the user does not correctly identify the framework packages. For instance, for the concept Context Menu, the instruction `setMenu()` (l. 209 in Figure 4) is missing because it is in the Eclipse framework, not in JFace. The only false negative for the concept Tree Viewer was the method `getChildren()`: since the collected traces included only the outgoing calls to this method and not any incoming calls, the generated template did not contain this method as one that needs to be implemented. Two of the false negatives for Moving Shapes were caused by a limitation of AspectJ, which cannot introduce code into `java.awt.*` packages or any other package belonging to the Java runtime library. The other two false negatives for this concept were due to different instructions that our sample applications used to change the location of a shape. Similarly, although we did not have any false negatives for Circle Drawing, the analysis revealed that one false negative was likely. The reason is that there are multiple ways of implementing circle drawing, e.g., using `drawOval()` or `draw(new Ellipse)`, and the difference between these calls is not visible to the application user.

Threats to Validity. We see three main threats to validity. First, the selection of frameworks and concepts for the evaluation might not be representative of those used in real-world development. This threat is addressed by selecting frameworks that are widely used, by including concepts from developer forums, and selecting concepts with different properties. Second, the selection of applications and execution traces might not be representative. We minimize this threat by following the same identification strategies

that would be applied in practice and use a mix of them in the evaluation. Third, the reference templates could be incorrect, which would impact the calculation of precision and recall values. We minimized this threat by (i) using three sources of knowledge for all concepts: manual inspection of sample applications, consulting existing documentation, and testing the implementation steps in sample implementations; (ii) 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 (iii) reporting not only the values for the comparison with the optional reference templates, but also to the mandatory reference templates.

5.2 Evaluation of Template Usage

Experimental Design. The previous experiment showed that implementation templates can be extracted from sample applications with high precision and recall. This experiment was designed to go further and evaluate the following research questions: (i) Are implementation templates as effective as framework documentation in aiding the development of framework-provided concepts? The rationale behind this question is that if templates are as effective as framework documentation, then they can serve as a substitute when no documentation is available. (ii) What is the influence of template quality and its usage strategies on the quality of resulting implementations? For instance, if templates are simply pasted into target applications, its false positives could pollute implementations with undesired code, and its false negatives could yield incomplete implementations.

Hypothesis and Measures. The experiment was designed to provide quantitative and qualitative evidence regarding the first research question, and qualitative data concerning the second question. Developers were recruited and asked to implement framework-provided concepts using one of the documentation aids, i.e., framework documentation or template. To answer the first research question, the effectiveness of each documentation aid was measured in terms of development time and functional correctness of resulting implementations. We defined three levels of functional correctness: *success*, if the resulting implementation behaved as specified; *buggy*, if the implementation did not perform as specified; and *incomplete*, if the developer did not finish the implementation. Since we had no expectations as to which documentation aid is superior we formulated the following two-sided null hypothesis:

H_0 : The development time to implement a framework-provided concept supported by implementation templates equals the development time when supported by framework documentation.

After the completion of the implementation, developers answered a questionnaire and went through a debriefing interview. To answer the second research question, this data was used to determine how developers employed the information contained in documentation and templates (usage strategies). We then analyzed how templates' quality and usage strategies affect the functional correctness of resulting implementations.

Procedure and Data Collection. We selected two simple concepts (Context Menu and Navigate) and two complex concepts (Content Assist and Table Viewer) that were used in the previous experiment (Table 1). We recruited twelve subjects: nine graduate students (S_1-S_6, S_8-S_{10}), two professionals (S_7, S_{12}), and one 4th-year undergraduate

student (S_{11}). Before the experiment, subjects answered a background questionnaire to determine their experience. All subjects had between 4 and 10 years of Java programming experience and all subjects except subject S_8 had at least one year of industrial programming experience. The subjects were blocked into two groups depending on their experience levels with the JFace and Eclipse frameworks: subjects in the *experienced* group (S_1-S_6) had implemented both frequent concepts before (Context Menu and Table Viewer), but not the rare ones (Navigate and Content Assist); subjects in the *moderate* group (S_7-S_{12}) had not previously implemented any of the four concepts. All subjects were given a briefing about implementation templates. In this briefing, we focused on the information available in templates, but let subjects freely decide how to use it in their implementations.

The experiment used three independent variables with two factor levels each: *documentation aid* (framework documentation (D) and implementation template (T)), *concept complexity* (simple and complex) and *subject experience* (moderate and experienced). The templates used in the experiment were those automatically extracted in the first study (Table 1). The documentation for a given concept was identified in Eclipse Help, Eclipse Corner Articles[1], or third-party Eclipse articles (web search). The documentation length varied between 5 pages (for Navigate) and 28 pages (for Content Assist). Each document had a dedicated section for the concept of interest and included code examples. Each subject was assigned one simple and one complex concept, and used a template for implementing one concept and documentation for the other concept (in a balanced sequence). The experienced subjects were assigned the rare concepts and the moderate subjects were assigned the frequent concepts; however, the assignment was random and balanced over the simple and complex concepts within each subject group. For each concept, a concept instance specification and a target application were created. Each specification consisted of a screenshot showing the desired concept to be implemented and a short paragraph describing it. Each target application was of minimal size, ranging between 10 (for Content Assist) and 186 LOC (for Navigate), to help developers focus on implementing the assigned concept instead of navigating and investigating the target applications.

The subjects were asked to implement the specified concept instance within the corresponding target application. During the implementation, the only documentation aids the subjects could use were the respective documentation aid (D or T), the two sample applications for a given concept (Table 1), and the framework-provided JavaDoc documentation. In particular, they could not use Eclipse Help or search the web. Note that JavaDoc documentation does not explain how to implement concepts, but only how to use a given framework-provided programming element, such as an interface or a method. The sizes of the sample applications varied between 1 KLOC (EditorList for Table Viewer) and 66 KLOC (Subclipse for Navigate). The subjects were instructed to implement each assigned concept without interruptions and measure the time needed.

After the implementation, subjects filled a result questionnaire, asking whether the provided documentation aid was useful; how many of the sample applications were used; and how the templates could be improved. Finally, we tested and inspected the subjects' code to determine whether each concept was correctly implemented.

[1] `http://www.eclipse.org/articles/`

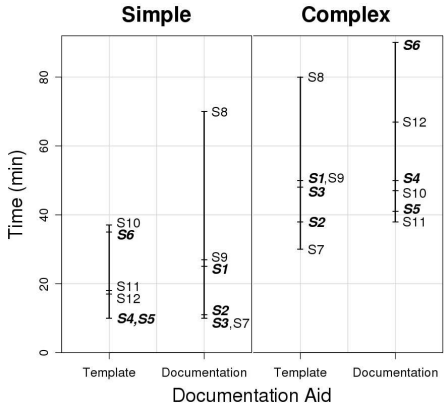

Fig. 7. Experimental results

Data Analysis. The qualitative aspects of the experiment were analyzed via inspection of resulting implementations, and careful examination of questionnaires and interviews. The quantitative assessment of development time was performed through univariate statistical analyses, which were applied to each independent variable. Unpaired, two-sample, two-sided *t-tests* [8] were performed. To reduce potential threats due to violation of t-test assumptions, we also applied the non-parametric Wilcoxon rank sum test [8]. For both hypothesis tests, we set the significance level to $\alpha = 0.05$ and presented the p-value. Furthermore, we measured the effect size as the percentage difference between means (*%diff*) and by calculating Hedge's *g* [9], which is defined as the difference between means divided by the pooled standard deviation.

Experimental Results. Due to space constraints, this section summarizes the most important results. A complete description of the data can be found elsewhere [6][10].

Discussion of Quantitative Results. Figure 7 shows the time measured for each implementation as a function of the documentation aid and concept complexity. Bold labels identify experienced subjects; solid lines indicate the variance. The descriptive statistics and the results of the statistical analyses are presented in Table 2. The three initial rows present data for the independent variables and the two last rows present additional data for documentation aid when isolating the complexity levels.

The first row presents the most important quantitative experimental results. The p-values for documentation aid are not significant ($p \gg 0.05$). On average, subjects using templates took 13% less time to implement the concept than subjects using documentation. The second row presents a different picture. The choice between a simple or complex concept has an extremely significant ($p = 0.0006 \ll 0.05$) impact on development time. Complex concepts take consistently longer than simple concepts to implement (avg. 124.6%), regardless of documentation aid and experience. This outcome inspired the analysis presented in the last two rows, where we isolate the complexity levels. The results show that, within a complexity level, subjects using templates were 11% or 17% faster on average, but these two results are statistically not significant, as in the first analysis. All these trends can also be clearly verified by inspecting the diagram in Fig-

Table 2. Descriptive statistics and univariate analysis results for development time

Independent Variable	Factor Level	Mean	Std. Dev.	%diff	Hedge's g	t-test (p-value)	Wilcoxon (p-value)
Documentation Aid (All Concepts)	Documentation	40.5	25.79	-13.0	-0.22	0.5855	0.6851
	Template	35.25	20.33				
Concept Complexity	Simple	23.33	17.71	124.6	1.58	0.0006	0.0006
	Complex	52.42	17.84				
Subject Experience	Moderate	40.92	22.49	-14.9	-0.25	0.5267	0.5431
	Experienced	34.83	23.82				
Documentation Aid (Simple Concepts)	Documentation	25.5	23.12	-17.0	-0.22	0.695	1.0000
	Template	21.17	11.99				
Documentation Aid (Complex Concepts)	Documentation	55.5	19.71	-11.1	-0.31	0.5748	0.7466
	Template	49.33	17				

ure 7, where complexity deeply impacts time, while documentation aid does not. Note that the diagram also allows the identification of individual trends (e.g., subject S_8 takes usually longer to perform the tasks).

The fact that H_0 could not be rejected with high significance in our experiment implies that no evidence could be provided regarding the difference (or equality) in effectiveness between templates and documentation when providing aid for developers to implement framework-provided concepts. The experiment shows that the impact of concept complexity is far greater than the choice of documentation aid, however, which indicates that if there is indeed difference, its impact on the development time is likely to be small.

Discussion of Qualitative Results. We first investigated the impact of templates' false positives and false negatives on the implementation. Table 1 shows the number of false positives and negatives for each template. In general, false negatives prevent the full instantiation of the concept, as in the case of Context Menu and Table Viewer. False positives either cause runtime errors (as in Content Assist) or pollute the concept instantiation with unnecessary code (as in Navigate). Further, the questionnaires and interviews allowed the determination of template usage strategies. Subjects used the templates essentially in two ways, either by pasting them directly into a target application or as an entry point to inspect sample applications. When using templates as an entry point, subjects either copied sample application code into the target application or wrote their own code based on what they learned. Some subjects who pasted templates directly into target applications also investigated sample applications before executing the code. Other subjects tried to execute the code and only investigated sample applications after a runtime error occurred or an unintended behavior appeared. Only subject S_4 pasted template code without verifying against sample applications.

The resulting implementations were executed and manually inspected. Most implementations followed the specified functional requirements and were classified as *success*. Only two were *buggy*. The task specification was not correctly followed by subject S_6 in the Content Assist (complex/documentation) implementation. Subject S_4 implemented Navigate (simple/template) with an additional button. This error arose because S_4 pasted the template code without verifying sample applications, and Navigate

contained false positives that included unwanted code. This observation suggests that templates should always be used together with sample applications since they help understanding what is missing and detecting unneeded code.

Threats to Validity. The main threat to internal validity concerns the distribution of subjects over the tasks. This threat was minimized by blocking subjects according to experience and randomizing the remaining distribution. The main threat to construct validity is related to the measurement of effectiveness and the definition of documentation. We used implementation time to measure effectiveness and it is clear that different notions could be used, such as code quality, that could affect the results. The definition of documentation sought to maximize its familiarity and conciseness by selecting standard documents dedicated to the concept at hand. The principal threats to external validity refer to the generalization from students to professionals and from a laboratory to a real setting and to the small sample size. We minimized these threats by selecting a sample that resembles our target population (i.e., experienced subjects) and using realistic concepts and a state-of-the-art development environment (Eclipse JDT). Our sample of twelve subjects is sufficient to produce preliminary results and qualitative insights, but a larger sample is required to provide conclusive results.

6 Discussion

Strengths and Weaknesses. The results of template extraction evaluation presented in Section 5.1 indicate that FUDA can retrieve concept implementation templates with relatively high precision and recall from only two traces and two sample applications. Furthermore, the processing of the traces is fully automatic and the instrumentation does not impose significant overhead on the application execution since only the API interaction rather than full traces are recorded. Given a set of applications and scenarios, the amount of time needed to retrieve templates is mainly determined by the time it takes to execute the scenarios on the applications. Furthermore, dynamic analysis detects the API elements that are actually being invoked. This is important since frameworks typically use polymorphism and reflection, which can render static analysis less precise.

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

Scenario Design Considerations. The nature of the concept and the ways in which it is implemented by the applications can influence the results. Ideally, the concept is atomic, its invocation is easily delimitable (for marking), and the sample applications have only this concept in common. In this case, FUDA will yield best results. In gen-

eral, concepts are composites of other concepts, the invocation of a concept might not be easily demarcated, and the sample applications may have several concepts in common. For a composite concept, developers should select applications that vary those of its components that should be eliminated. If the concept of interest is part of a composite concept, developers should be able to demarcate the boundaries of the concept execution. If these scenario design goals are only partially satisfied, the resulting false positives and false negatives may still be identified by following the traceability links in the template and studying the actual sample application code.

7 Related Work

Framework documentation and completion approaches support framework users passively or actively or both. For instance, *framework-specific modeling languages* (FSMLs) [11] document framework-provided concepts as hierarchies of mandatory and optional features and actively support users in instantiating the concepts through round-trip engineering. Further, *reuse contracts* [12] and *collaboration contracts* [13] help ensure that frameworks are used correctly. Nonetheless, the main difficulty of these approaches is that framework documentation requires manual effort and, consequently, documentation of the framework may become outdated. FUDA attempts to fill this gap by allowing users to generate implementation templates when the framework documentation is missing.

Framework usage comprehension is supported by several approaches such as XSnippet [14], Strathcona [2], Prospector [15], PARSEWeb [16], and FrUiT [3]. Both XSnippet and Strathcona are context-sensitive code assistant tools that allow developers to query a repository of code snippets that are relevant to the programming task at hand. Given two API types τ_{in} and τ_{out} as a query, both Prospector and PARSEWeb mine for call sequences transforming an object of type τ_{in} to another object of type τ_{out}. FrUiT mines for frequent API usage patterns as association rules, e.g., *subclass A* $\Rightarrow$ *call m*. It then uses such rules to suggest implementation steps for a class under development. All these approaches are mainly code assistants in the context of a programming task at hand, such as how to call a specific framework method or how to instantiate a particular framework class. In contrast to FUDA, they do not provide a complete code snippet or implementation template for instantiating an entire, large concept, which may span multiple framework methods or even classes. Moreover, whereas all these approaches use static analysis, FUDA applies primarily dynamic analysis. The advantage of static analysis is that it can cope with a large body of applications and potentially incomplete code. The advantage of dynamic analysis is that it can handle highly polymorphic and reflective code, which is often part of modern frameworks. Additionally, contrary to FUDA, static analysis does not support concept identification by invoking concepts directly from the user interface.

Specification mining is concerned with automatically discovering the protocols or rules that a program must follow when interacting with an API. Existing techniques can be classified into static [17][18] and dynamic [19][20][21] ones. Examples of static approaches include inferring ordering patterns among method calls [17] or detecting function precedence protocols [18]. Examples of dynamic approaches contain mining

temporal API rules from dynamic traces [19], mining iterative patterns from traces [20], or inferring declarative specifications of the API behavior for target concepts such as the raising of an exception [21]. In contrast to specification mining approaches, FUDA does not recover API interaction protocols. The latter are important for library API usage, but less so for frameworks. Frameworks typically follow *inversion of control* by enforcing protocols in framework rather than application code. Although FUDA extracts total orders of calls, it does so to improve readability of templates by sorting calls within method bodies. Additionally, dynamic specification mining techniques often require several runtime traces in order to recover different legal execution sequences. On the other hand, FUDA aims to keep the number of traces as small as possible to make the approach attractive in practice.

Concept location concentrates on understanding how a certain concept or functionality is implemented in the source code of an application. Existing approaches can be mainly categorized into static (e.g., [4]), dynamic (e.g., [22]), and hybrid ones (e.g., [5][23]). One can refer to [5] for a good literature overview. We focus only on the most related dynamic and hybrid techniques. Most of these techniques use two or more traces to filter out irrelevant events, e.g., [22][23]. SITIR [5] gets away with only one trace by filtering it using the textual similarity to a keyword query. Unlike FUDA, all these techniques focus on retrieving concepts in general application code rather than framework-provided concepts. Therefore, the result may contain many application-specific instructions that are irrelevant from the viewpoint of framework usage. FUDA avoids this problem by focusing on API interaction traces and removing the application-specific content from those traces through the event generalization. Furthermore, we are unaware of other techniques using the combination of API trace marking with API trace slicing. In particular, SITIR [5] uses the runtime trace marking to reduce the size of the traces, but it misses the relevant events to the implementation of the desired concept that are not marked at runtime. FUDA is able to identify such relevant events by applying the API trace slicing.

8 Conclusion

This paper presented FUDA, an approach for extracting implementation templates from traces obtained by invoking concepts of interest in sample applications. FUDA was tested on twelve concepts of four widely-used frameworks. The concept sample included both simple and complex ones. Six concepts corresponded to questions found at developer forums. The experimental evaluation shows that, for the considered concepts, FUDA can extract templates with high precision (59-100%) and recall (79-100%) from only two traces and two sample applications per concept. Finally, we reported on a user experiment with twelve subjects in which the choice of templates vs. documentation had much less impact on development time than the concept complexity. The experiment also suggested that the templates should be used together with the sample applications from which they were extracted.

Acknowledgements. The authors would like to acknowledge Eric Eide, James Noble, and the anonymous reviewers for their valuable suggestions for improving the paper. We also thank Shoja Chenouri who helped in the statistical analyses.

References

1. Gamma, E., Beck, K.: Contributing to Eclipse: Principles, Patterns, and Plugins. Addison-Wesley (2003)
2. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: ICSE. (2005)
3. Bruch, M., Schafer, T., Mezini, M.: FrUiT: IDE support for framework understanding. In: ETX. (2006)
4. Zhao, W., Zhang, L., Liu, Y., Sun, J., Yang, F.: SNIAFL: Towards a static noninteractive approach to feature location. TOSEM **15**(2) (2006)
5. Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.: Feature location via information retrieval based filtering of a single scenario execution trace. In: ASE. (2007)
6. Heydarnoori, A.: Supporting Framework Use via Automatically Extracted Concept-Implementation Templates. PhD thesis, University of Waterloo, Canada (February 2009)
7. Heydarnoori, A., Bartolomei, T.T., Czarnecki, K.: Comprehending object-oriented software frameworks API through dynamic analysis. Technical Report CS-2007-18, School of Computer Science, University of Waterloo (2007)
8. Montgomery, D.C.: Design and analysis of experiments. 6th edn. Wiley (2004)
9. Hedges, L.V.: Distribution theory for Glass's estimator of effect size and related estimators. Journal of Educational Statistics **6**(2) (1981)
10. Generative Software Development Lab: FUDA supporting material. `http://gsd.uwaterloo.ca/~aheydarn/fuda/` (2008)
11. Antkiewicz, M., Bartolomei, T.T., Czarnecki, K.: Automatic extraction of framework-specific models from framework-based application code. In: ASE. (2007)
12. Steyaert, P., Lucas, C., Mens, K., D'Hondt, T.: Reuse contracts: managing the evolution of reusable assets. In: OOPSLA. (1996)
13. Hondt, K.D.: A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems. PhD thesis, Vrije Universiteit Brussel, Belgium (1998)
14. Sahavechaphan, N., Claypool, K.: XSnippet: Mining for sample code. In: OOPSLA. (2006)
15. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: Helping to navigate the API jungle. In: PLDI. (2005)
16. Thummalapenta, S., Xie, T.: PARSEWeb: A programmer assistant for reusing open source code on the web. In: ASE. (2007)
17. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: FSE. (2007)
18. Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: ICSE. (2007)
19. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: Mining temporal API rules from imperfect traces. In: ICSE. (2006)
20. Lo, D., Khoo, S.C., Liu, C.: Efficient mining of iterative patterns for software specification discovery. In: KDD. (2007)
21. Sankaranarayanan, S., Ivanči, F., Gupta, A.: Mining library specifications using inductive logic programming. In: ICSE. (2008)
22. Wilde, N., Scully, M.C.: Software reconnaissance: Mapping program features to code. JSM **7**(1) (1995)
23. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. TSE **29**(3) (2003)