Model-guided Code Assistance for Framework Application Development

by

Hon Man Lee

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2009

Declaration

Abstract

Object-oriented frameworks are currently widely used in software application development. Unfortunately, they are known to be generally difficult to use because of the difficulty in understanding the concepts and constraints in different frameworks. With the formalization of framework concepts and constraints in domain-specific modeling languages called framework-specific modeling languages (FSMLs), previous works have shown that round-trip engineering between models of applications using frameworks and the application code is possible to aid framework application development.

Framework-specific modeling languages only capture, however, framework concepts and constraints and hence, lack the expressiveness of general-purpose modeling languages. For this reason, the complete code for an entire framework application cannot be generated from the model in the model editor using round-trip engineering, and the user would need to switch to the code editor to program the application logic code. Also, since models are only abstractions of code, implementation details in code may be missing in models. Although default implementation details can be used when generating code from a model, the generated code might require further customization by the user, which would also require switching to the code editor.

To reduce the need for the user to switch between the model editor and the code editor and to reduce the need to customize the generated code, this thesis presents a model-guided approach to providing code assistance for framework application development directly in the code editor, where additional implementation details can also be obtained. An approach to building a context-sensitive code assistant that aids the user in the implementation of framework concepts with the consideration of framework constraints is described. A prototype has further been implemented and applied on two widely popular frameworks. The evaluation in this thesis analyzes and characterizes framework concepts and shows that the framework-based code assistant can reduce the need to customize the generated code in the code editor when compared to code generation from the model editor.

Acknowledgments

I begin by thanking my supervisor, Professor Krzysztof Czarnecki, for accepting me as his student and guiding me through the journey of research. I am also thankful for his advice on not only my research, but also on other matters that has helped me to develop personally.

I must also thank Michal Antkiewicz, who is a colleague of mine in the Generative Software Development Lab and whom I have collaborated with for the research presented in this thesis. His patience in introducing me to his doctoral work on framework-specific modeling languages and its support for round-trip engineering, which this research builds upon, is much appreciated.

I would also like to thank my other colleagues in the Generative Software Development Lab. They were supportive of me and provided me with an encouraging and enjoyable environment to work in.

My thanks also goes to my thesis committee members, Professor Ric Holt and Professor Patrick Lam, for taking the time out of their busy schedules to read my thesis and to provide valuable comments on it.

Finally, I would also like to thank my family for their unconditional love and support for me.

Contents

List of Tables

List of Figures

Chapter 1

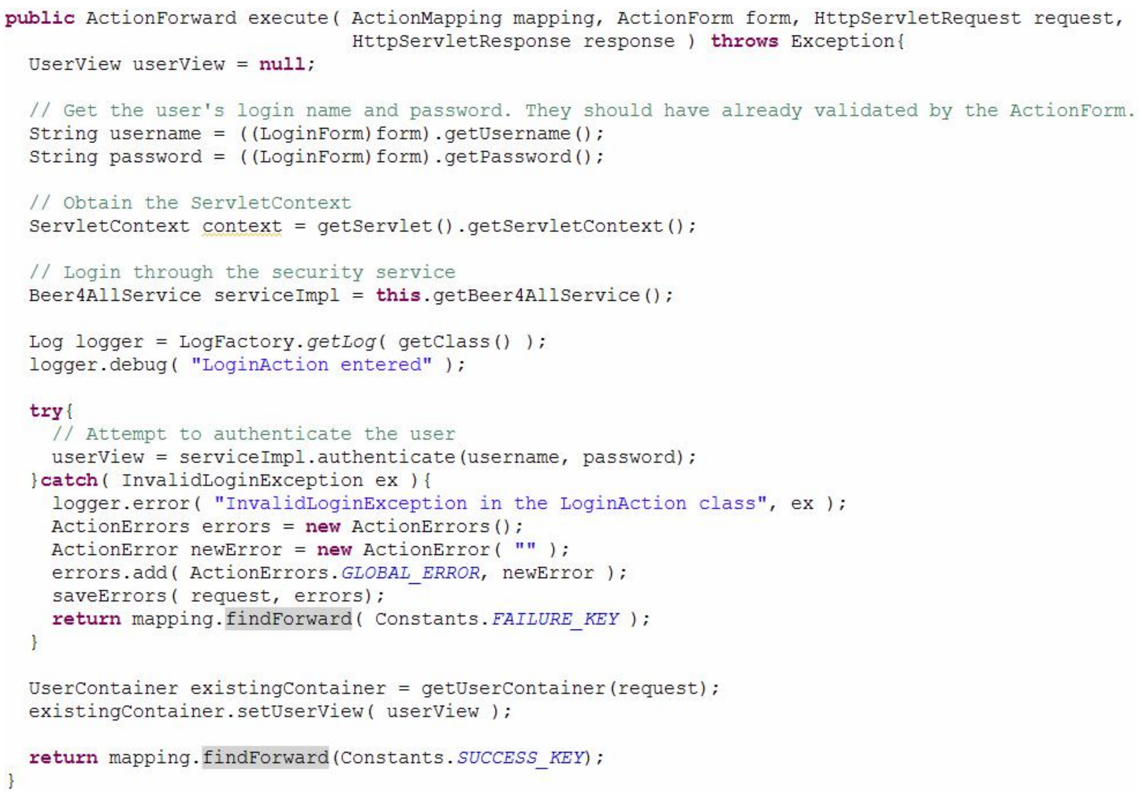
Introduction

Object-oriented frameworks are currently used in many domains for software application development. A framework implements concepts in a specific domain that applications can instantiate, in a process often known as *framework instantiation*, and defines constraints that applications must follow when instantiating the concepts. For example, version 1.x of the Apache Struts Java web framework implements concepts such as *forms*, *actions* and *forwards*. Forms store user inputs, and actions process the inputs, returning forwards that redirect to actions or web pages. To instantiate the concept `forward` in an action, the framework application programming interface (API) for Struts provides the method findForward(String) in the ActionMapping class, enabling a method call to findForward(String) on ActionMapping objects in Action classes. A constraint that is imposed by the framework on the instantiation of forward is that the value of the parameter in the method call must be either a local forward or a global forward defined in a Struts XML configuration file. A local forward only works for the Action class that it is defined for whereas a global forward works for all Action classes.

Despite their popularity, frameworks are notoriously difficult to use [Hou et al., 2005, Kirk et al., 2007]. Applications that instantiate frameworks are required to use many different mechanisms to instantiate the concepts defined in frameworks, such as extending framework API classes, implementing framework API interfaces, and making framework API calls [Antkiewicz et al., 2008]. Framework concepts are sometimes tightly coupled or delocalized and are often not well documented [Hou et al., 2005]. Kirk et al.'s empirical study on framework reuse [Kirk et al., 2007] identified four major problems with the use of frameworks: understanding the functionality of framework components, understanding the interactions between framework components, understanding the mapping from the problem domain to the framework implementation, and understanding the architectural assumptions in the framework design.

Framework-specific modeling languages (FSMLs) [Antkiewicz and Czarnecki, 2006, Antkiewicz, 2008] were proposed by Michal Antkiewicz to formalize the concepts and constraints in different frameworks. FSMLs are domain-specific modeling languages that can be used to express framework-specific models, which are models that describe framework concepts instantiated in application code. Michal further demonstrated the feasibility of round-trip engineering, which involves reverse engineering, forward engineering, and synchronization, with the Java applet framework and a subset of the Eclipse framework that captures the interaction between Eclipse workbench parts [Antkiewicz, 2008]. Reverse engineering extracts framework-specific models from application code, forward engineering generates application code from framework-specific models, and synchronization synchronizes changes between models and code. Instead of merely overwriting the existing model in reverse engineering and the existing code in forward engineering, synchronization allows both the model and the code to be independently modified and synchronized through incremental updates [Sendall and Küster, 2004]. Unlike most current commercial tools that only support round-trip engineering between UML class diagrams and code and hence, generating only class frames and method declarations without bodies, round-trip engineering for FSMLs generates functional code of framework concepts captured by FSMLs, including method calls inside method bodies.

While reverse engineering involves the extraction of an abstract representation of the code from the code itself, forward engineering requires the generation of code from an abstract representation of the code that may lack some implementation details. For example, Figure 1.1 is an example of a code snippet from a Struts application. The example demonstrates that, to generate the method call



```
public ActionForward execute( ActionMapping mapping, ActionForm form, HttpServletRequest request,
                              HttpServletResponse response ) throws Exception{
  UserView userView = null;

  // Get the user's login name and password. They should have already validated by the ActionForm.
  String username = ((LoginForm)form).getUsername();
  String password = ((LoginForm)form).getPassword();

  // Obtain the ServletContext
  ServletContext context = getServlet().getServletContext();

  // Login through the security service
  Beer4AllService serviceImpl = this.getBeer4AllService();

  Log logger = LogFactory.getLog( getClass() );
  logger.debug( "LoginAction entered" );

  try{
    // Attempt to authenticate the user
    userView = serviceImpl.authenticate(username, password);
  }catch( InvalidLoginException ex ){
    logger.error( "InvalidLoginException in the LoginAction class", ex );
    ActionErrors errors = new ActionErrors();
    ActionError newError = new ActionError( "" );
    errors.add( ActionErrors.GLOBAL_ERROR, newError );
    saveErrors( request, errors);
    return mapping.findForward( Constants.FAILURE_KEY );
  }

  UserContainer existingContainer = getUserContainer(request);
  existingContainer.setUserView( userView );

  return mapping.findForward(Constants.SUCCESS_KEY);
}
```

Figure 1.1: Framework-specific concepts intertwined with application logic

`findForward(String)` for the concept `forwards`, besides knowing the class and method to generate the method call in, forward engineering needs to know the exact line in the method body to place the method call in. The example also demonstrates how a FSML concept might be completely intertwined into the application logic code, which is not captured by the framework-specific model since framework-specific models only capture instantiations of framework concepts. In the example, depending on whether the user authenticates successfully or not, different web pages are returned as forwards.

The current implementation for forward engineering attempts to generate code at fixed location predefined in the FSMLs, such as in the last line of the method body in the previous example, and completely ignores the user's existing code. After adding new concepts in the framework-specific model in the model editor, the implementation relies on the user to manually synchronize the changes between the model and the code to generate code and to manually customize the generated code in the code editor. Unlike general purpose modeling languages, since FSMLs are domain-specific modeling languages that only cover selected framework concepts, working in the code editor is necessary to program application logic that is outside the scope of the FSMLs. The frequent switching between model editor and code editor and the need to customize the generated code interrupts the flow of development and impedes the practicability of round-trip engineering with FSMLs.

To bridge the abstraction gap between model and code and to reduce the interruption in the flow of development from switching between the model editor and the code editor, this thesis proposes an approach of providing code assistance directly in the code editor that is guided by framework-specific models of the code extracted using reverse engineering. We describe a framework-based code assistant that provides context-sensitive suggestions to implementing framework concepts with the consideration of framework constraints. The framework-based code assistant can generate code from the model as in forward engineering, but can be invoked in the code editor directly and automatically synchronizes code changes with the model. Also, the code assistant allows for code to be generated at varying location and provides the code context information that is required in code generation. We have also implemented a prototype to demonstrate the feasibility of the approach and has applied the prototype on two widely popular frameworks, the Apache Struts framework, which has a corresponding FSML that only supported reverse engineering previously, and the Java applet framework. The evaluation in this thesis tests the hypothesis that the framework-based code assistant approach reduces the effort required to produce framework application code compared to forward engineering from the model editor. The evaluation focuses on examining the differences between the two approaches in their need to customize the generated code by moving the code to the intended location. We first classify the variability in location in instances of framework concepts in code and then, we manually inspect the code to understand the extent to which instances of framework concepts entangle with application logic code.

1.1 Research Contributions

The key contributions in this thesis are:

- A model-guided approach to providing advanced context-sensitive code assistance to aid the implementation of framework concepts with the consideration of framework constraints
- A prototype implementation of the approach and its application to two FSMLs to support two widely popular frameworks
- Analysis and characterization of every feature in the two FSMLs to show that framework-based code assistance lowers developers' effort to produce code in framework applications when compared to forward engineering from the model editor

1.2 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 describes related work, first surveying code assistants found in today's IDEs and then surveying other related work in the research literature.
- Chapter 3 reviews key concepts about framework-specific modeling languages applicable to creating framework-specific models that guide the proposed framework-based code assistant. The chapter also briefly reviews round-trip engineering with framework-specific models, which our framework-based code assistant relies on.
- Chapter 4 presents the framework-based code assistant. The chapter first describes a general methodology of creating the code assistant, after which a description of a prototype implementation of the code assistant is presented. Finally, the chapter walks-through two examples of using the prototype on two applications, each using a different framework.
- Chapter 5 evaluates the framework-based code assistant by analyzing every feature in two FSMLs and comparing the effort required to produce framework application code using the code assistant with forward engineering from the model editor previously supported in round-trip engineering with FSMLs.
- Chapter 6 concludes this thesis and presents a list of possible directions for future work.

Chapter 2

Related Work

This chapter first examines code assistants in current integrated development environments (IDEs) and then surveys the literature and compares the approaches in the literature with the approach proposed in this thesis.

2.1 Code Assistants in Current Integrated Development Environments

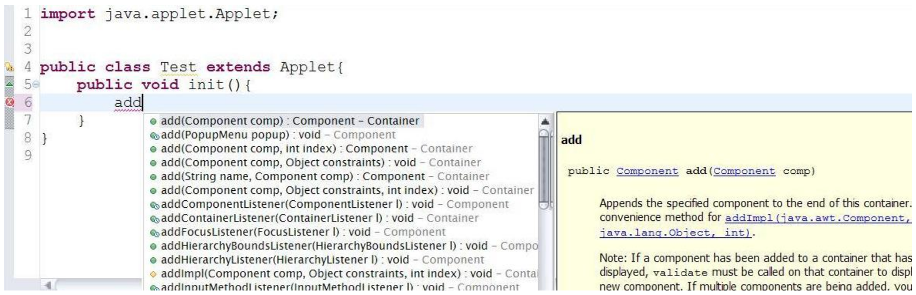
This section gives an overview of three types of code assistants found in current IDEs: autocomplete, error identification with quick fixes, and quick assists.

2.1.1 Autocompletion

By far the most common code assistant in current IDEs is autocompletion. Much like the autocompletion found in address bars and web forms in most modern web browsers, or fields in dialog boxes in most modern operating systems, autocompletion in IDEs attempts to complete the typing the user has started, which in this case is code in the source code editor. Several names for autocompletion have been used in different IDEs. For example, autocompletion is known as IntelliSense in Microsoft Visual Studio and as content assist in Eclipse. The following list is a list of capabilities of Content Assist in Eclipse, adapted from [Carlson, 2005]:

- Complete type, variable, or method names anywhere in your code
- Guess new variable or parameter names from their types
- Insert code templates representing common coding patterns
- In Javadoc comments, insert HTML tags or standard Javadoc tags
- Fill field values in dialogs and wizards

A study done recently by Murphy et al. [Murphy et al., 2006] on how Java software developers use the Eclipse IDE reports that content assist in Eclipse is the fifth most-used command in the IDE, ranking it among common editing commands such as copy and paste. Indeed, autocompletion is a command that software developers heavily rely on in software development. Besides alleviating the burden of remembering long method signatures in APIs or programming keywords by automatically completing prefixes entered by the user, it is often used as documentation browser. It is a common practice for developers to invoke autocompletion without entering a prefix and then scroll through the complete list of method calls suggested by autocompletion to identify the method call that matches the user's intent. In fact, both Eclipse and Visual Studio automatically trigger autocompletion when the user types in ".", "->", or "::" after a receiver object, proposing the complete list of possible method calls from the receiver object type's API. Contrary to the case when the prefix of the desired method call is entered before invoking autocompletion, in this case, the user usually does not have in mind the particular method call desired when invoking autocompletion. We distinguish these two modes of using autocompletion by calling the mode where the prefix of the target code is first entered before invoking autocompletion *completion-based autocompletion* and the mode where no prefix is entered before invoking autocompletion *suggestion-based autocompletion*, usually as a mean of documentation. Figure 2.1(a) shows an example of completion-based autocompletion in Eclipse and Figure 2.1(b) shows an example of suggestion-based autocompletion.



```
import java.applet.Applet;


public class Test extends Applet{
    public void init(){
        add
    }
}

```

```
add(Component comp) : Component – Container
add(PopupMenu popup) : void – Component
add(Component comp, int index) : Component – Container
add(Component comp, Object constraints) : void – Container
add(String name, Component comp) : Component – Container
add(Component comp, Object constraints, int index) : void – Container
addComponentListener(ComponentListener l) : void – Component
addContainerListener(ContainerListener l) : void – Container
addFocusListener(FocusListener l) : void – Component
addHierarchyBoundsListener(HierarchyBoundsListener l) : void – Compo
addHierarchyListener(HierarchyListener l) : void – Component
addImpl(Component comp, Object constraints, int index) : void – Contai
addInputMethodListener(InputMethodListener l) : void – Component
```

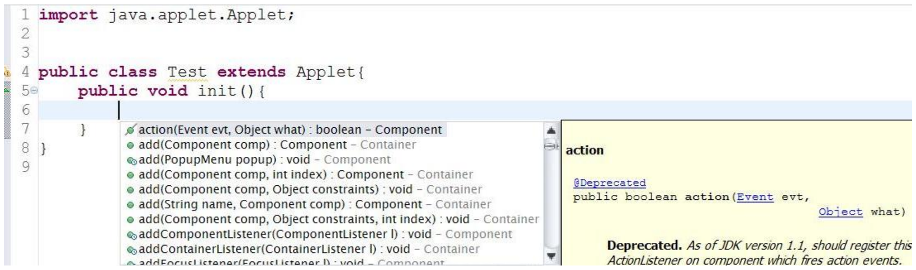
add

`public Component add(Component comp)`

Appends the specified component to the end of this container.
convenience method for `addImpl(java.awt.Component, java.lang.Object, int)`.

Note: If a component has been added to a container that has
displayed, `validate` must be called on that container to displ
new component. If multiple components are being added, you

(a) Eclipse Completion-based Autocompletion



```
import java.applet.Applet;


public class Test extends Applet{
    public void init(){
        
    }
}

```

```
action(Event evt, Object what) : boolean – Component
add(Component comp) : Component – Container
add(PopupMenu popup) : void – Component
add(Component comp, int index) : Component – Container
add(Component comp, Object constraints) : void – Container
add(String name, Component comp) : Component – Container
add(Component comp, Object constraints, int index) : void – Container
addComponentListener(ComponentListener l) : void – Component
addContainerListener(ContainerListener l) : void – Container
addFocusListener(FocusListener l) : void – Component
```

action

```
@Deprecated
public boolean action(Event evt,
                      Object what)
```

Deprecated. *As of JDK version 1.1, should register this ActionListener on component which fires action events.*

(b) Eclipse Suggestion-based Autocompletion

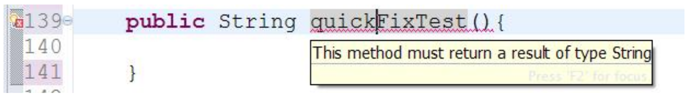
Figure 2.1: An example of the two modes of Eclipse content assist

Eclipse also provides content assist support in its XML editor through the Eclipse web standard tools project. However, the suggestions are based purely on the XML schema and are not framework-specific.

A recent version of Eclipse, version 3.4, has added support for code templates in its content assist that is similar to the framework-based code assistant presented in this thesis for the user. The difference, however, is that our approach is guided by a model, and the user can easily add support for a new framework by specifying the framework concepts and constraints declaratively with a new FSML. In the Eclipse approach, although new code templates can be specified in an XML file, no constraint checking is supported and only a limited set of context is supported. Also, the code templates in Eclipse does not allow cross-cutting code to be generated, such as code generated in both Java and XML files when maintaining a framework's referential integrity constraint.

2.1.2 Error Identification with Quick Fixes

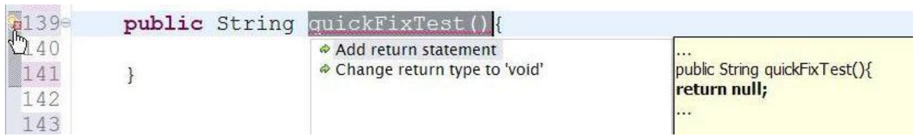
Another common code assistant found in current IDEs is code error identification. As in autocompletion, similar ideas are also easily found in everyday applications. For example, in word processors, as the user types, spell checking runs continuously in the background, and words with spelling mistakes are automatically underlined. In IDEs, as the user programs, code building runs continuously in the background, and code with compilation errors are automatically underlined in the code editor. In addition to code error identification, Eclipse further offers common fixes for the code errors identified, called quick fixes, which is analogous to the spelling suggestions in word processors. For example, when a return statement is missing in the body of a method with a return type of String, the method name is underlined with the error message "this method must return a result of type String" and the quick fix mechanism proposes the following fixes: 1) adding the statement *return null;* to the body of the method or 2) change the return type of the method to "void". Figure 2.2(a) is a screenshot of the example described for error identification and Figure 2.2(b) for quick fix. Readers are referred to [Ecl, 2007b] for a complete list of supported quick fixes in Eclipse.



```
public String quickFixTest(){

}
```

This method must return a result of type String

(a) Eclipse Error Identification



```
public String quickFixTest(){

}
```

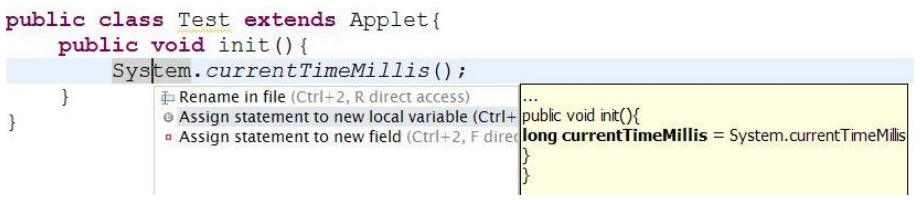
```
...
public String quickFixTest(){
return null;
...
```

(b) Eclipse Quick Fix

Figure 2.2: An example of Eclipse error identification and quick fix

2.1.3 Quick Assist

Similar to quick fixes, Eclipse also offers what is called quick assist. Quick assist is a small set of actions, much like quick fixes, but not necessarily for code with compilation errors. Instead, the actions are a set of common code transformations that programmers may perform, such as converting from a set of switch and case statements to a set of if-else statements or assigning an expression to a variable. Figure 2.3 shows a screenshot of quick assist in action, suggesting the assignment of an expression to a variable. Refer to [Ecl, 2007a] for a complete list of supported quick assists in Eclipse.



```
public class Test extends Applet{
    public void init(){
        System.currentTimeMillis();
    }
}
```

Rename in file (Ctrl+2, R direct access)
Assign statement to new local variable (Ctrl+
Assign statement to new field (Ctrl+2, F direc

```
...
public void init(){
long currentTimeMillis = System.currentTimeMillis
}
}
```

Figure 2.3: An example of Eclipse quick assist

2.2 Literature Survey

Related work in the literature can be broadly grouped into several categories: code generation, round-trip engineering, mining-based code assistants, improvements on autocompletion in traditional IDEs, and generation of domain-specific IDEs. Each of these categories is described in the subsections below.

2.2.1 Code Generation

In general, the generation of code from model can be seen as a type of model-to-code transformation, which can be classified into visitor-based approaches and template-based approaches [Czarnecki and Helsen, 2003]. Visitor-based approaches traverse the internal representation of a model to generate code. On the other hand, template-based approaches use code templates and control structures in the templates to traverse the model and expand the parameters in the templates. Most code generation tools on the market and in the literature uses the template-based approach. Examples of such tools include Java Emitting Templates (JET) [Ecl, 2007], Xpand in openArchitectureWare(oAW) [ope, 2008], and XDoclet [XDo, 2005]. On the other hand, forward engineering in FSMLs, which the proposed framework-based code assistant works on, uses the visitor-based approach. Forward engineering with FSMLs does not impose a strict structure on the generated code and can be used on code manually written. Also, it does not require mechanisms such as protected regions and partial classes, which are currently commonly used to protect changes to the generated code in the template-based approach. The tradeoff is that code templates are easier to write than code transformations from models. Since the proposed framework-based code assistant is intended to be used in any code, including code that is not generated using the code assistant, a visitor-based approach is more appropriate than the code template approach.

2.2.2 Round-trip Engineering

A number of commercial computer-aided software engineering (CASE) tools on the market support round-trip engineering between UML class diagrams and code. These tools only create class frames and method declarations without method bodies. One effort towards round-trip engineering of executable code is FUJABA [Nickel et al., 2000]. Unlike round-trip engineering with FSML, which only models framework concepts, FUJABA uses a high-level visual programming language called story diagrams that combines UML class diagrams and UML behavioral diagrams and supports round-trip engineering to produce complete executable Java code. As a result, FUJABA does not have the problem of integrating generated code with manually written code as all the code is captured in the story diagrams. With the manually written code in the story diagrams however, the story diagrams become more complicated and more difficult to understand. The difference between FUJABA and round-trip engineering with FSMLs is that FSMLs are domain-specific modeling languages that only capture domain-specific concepts whereas FUJABA's story diagrams are based on UML, which is a general-purpose modeling language that works on a lower level of abstractions than domain-specific modeling languages.

There is also a large body of theoretical work in the literature on the theories of round-trip engineering and bidirectional transformation in general [Sendall and Küster, 2004, Antkiewicz and Czarnecki, 2007, Foster et al., 2005, Stevens, 2007, Xiong et al., 2007]. Among them, [Antkiewicz and Czarnecki, 2007] discusses the need for a decision function as an additional input to select one input from a set of possible targets. In this sense, framework-based code assistant can be seen as a decision function in the context of forward engineering in round-trip engineering.

2.2.3 Improvements on Autocompletion in Traditional Integrated Development Environments

There are several efforts in the literature that aim to improve the traditional autocompletion found in current IDEs.

Keyword programming in Java [Little and Miller, 2007] allows users to input keywords in the code editor, which are then translated to a list of possible valid Java expressions in the current context. While we have also adapted the keyword programming approach in the framework-based code assistant presented in this thesis, keyword programming in Java only generates single method calls, whereas the framework-based code assistant described in this thesis is designed for frameworks and hence, supports a variety of mechanisms that frameworks use to implement abstractions, potentially involving multiple instructions code that are cross-cutting.

Robbes et al. [Robbes and Lanza, 2008] investigated the problem of sorting proposals, which are typically alphabetically sorted, in the traditional autocompletion. They used program history to improve the ranking of proposals and presented a test to evaluate different algorithms for ranking proposals. While we also consider ranking of proposals in this thesis, the focus of our research is on raising the level of abstraction in these proposals such that code for framework instantiation can be generated.

Mylyn [Kersten and Murphy, 2005] introduced the concept of task context to reduce information overload in IDEs by filtering and ranking information presented to the user using the user's interaction history with the IDE. Similar to [Robbes and Lanza, 2008], it also attempts to rank autocompletion proposals such that more relevant proposals are ranked higher. Our approach also ranks proposals and reduces information overload by grouping programming language constructs into framework concepts.

2.2.4 Mining-Based Code Assistants

There is a large body of work on code assistants based on mining on existing sample applications.

Strathcona [Holmes and Murphy, 2005] allows users to request examples for a class, a method, or a field declaration during framework instantiation based on the current code context. The tool first extracts structural information, such as the parent class of each class and the type of fields in each class, of sample applications on a server. It then matches the structural information in the current code in the code editor with those on the server based on six heuristics. It finally returns code examples that occur most frequently in the set after applying the heuristics.

FrUiT [Bruch et al., 2006] also aims to suggest a set of relevant code from sample applications for framework instantiation. However, they use association rule mining to mine for associations between the structural information in the current context in the code editor and the sample applications on the server.

XSnippet [Sahavechaphan, 2006] also attempts to suggest code by mining sample applications. However, their approach is targeted towards finding code snippets that are relevant to object instantiation.

Jungloid Mining (Prospector) [Mandelin et al., 2005] also mines in sample applications to suggest code but is targeted to the problem of finding example code that transforms an object of an input type to an object of an output type.

Automatic Method Completion [Hill and Rideout, 2004] attempts to automatically complete the current method the user is at in the code editor. The tool precomputes a 154-dimensional vector of metrics for each method in a set of sample applications on a server. It then uses the K-Nearest Neighbor data mining algorithm to compare the vector of metrics of the current method with the precomputed metrics on the server to find a relevant method that is similar to and longer than the current method.

Compared to these tools, the framework-based code assistant described in this thesis is based on a language-oriented approach that formalizes framework concepts and constraints using domain-specific modeling instead of mining from sample applications. The domain-specific modeling approach avoids inaccuracy inherited in data mining techniques by always mapping the current code to a framework-specific model and suggests and generates code based on interpreting the metamodel of the framework-specific model.

2.2.5 Generation of Domain-Specific Integration Development Environments

The creation of language programming tools and IDEs based on programming language description has long been the subject of research, dating back to the 80's [Reps and Teitelbaum, 1984]. Of these, recent efforts include Eclipse IDE Metatooling Platform (IMP) [Charles et al., 2007], TOPCASED [Farail et al., 2006], and Textual Generic Editor (TGE) [ATLAS, 2008]. Compared to the code assistant described in this thesis that supports multiple proposals to aid the implementation of a framework concept and generates cross-cutting code, the autocompletion generated for the programming environments in these works are more primitive, typically supporting only keywords in the language. There are also works on generating programming environments specific to frameworks [Bjarnason and Hedin, 1997, Hakala et al., 2001], but framework-based code assistants are not discussed in these works.

Chapter 3

Review of Framework-Specific Modeling Languages and Round-trip Engineering with Framework-Specific Models

This chapter starts with an overview of framework-specific modeling languages, which are used to express framework-specific models that the framework-based code assistant in the next chapter interprets. The chapter then provides an overview of round-trip engineering with framework-specific modeling languages.

3.1 Framework-Specific Modeling Languages & Framework-Specific Models

A framework-specific modeling language (FSML) [Antkiewicz, 2008, Antkiewicz and Czarnecki, 2006] is a domain-specific modeling language (DSML) designed for a particular framework and formalizes that framework's API concepts and constraints. FSMLs are used to express *framework-specific models*. A framework-specific model models how an application uses a framework's API by describing instances of concepts from the framework's FSML that are implemented by the application. Figure 3.1 describes the relationship of FSML, framework-specific models, frameworks, and applications.

In FSMLs, concepts are formalized as a cardinality-based feature model [Czarnecki et al., 2004]. Feature modeling is a technique for modeling commonality and variability in domain analysis [Kang et al., 1990], software product lines [Weiss and Lai, 1999, Clements and Northrop, 2001], and generative programming [Czarnecki and Eisenecker, 2000]. Since frameworks are designed to be highly extensible and reusable, feature models can be used to model framework concepts and constraints.

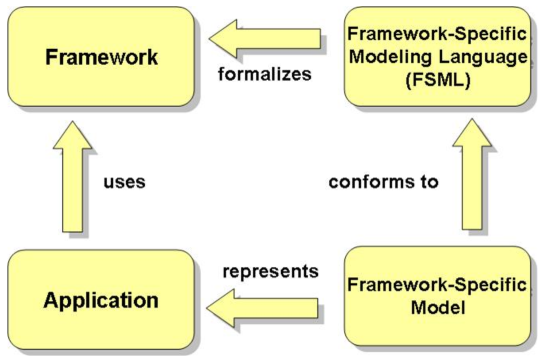


Figure 3.1: Overview of framework application modeling

In a feature model, a concept is decomposed into a hierarchy of features. Each feature is a property of a concept and has a cardinality constraint attached to it. The cardinality constraint specifies the number of instances of the feature that should exist in any configuration of the feature model. Features can also be organized into feature groups that have group cardinalities. A feature in a FSML can have a *mapping definition* attached, which specifies how the feature can be implemented and located in the code. Mapping definition for a feature is defined by assigning concrete values to parameters in one of the reusable mapping types. Mapping definition can also capture referential integrity constraints. These constraints are constraints on the consistency between two source artifacts, such as Java and XML, in a FSML. A complete list of supported mapping types and constraints and their descriptions from [Antkiewicz, 2008] is reproduced in Appendix A for convenience. Also refer to [Antkiewicz et al., 2009] for a description of a method of engineering new FSMLs. Semantically, a feature model describes a set of legal configuration of the features, known as *feature configurations*. A framework-specific model is, hence, a feature configuration that describes how an application implements framework concepts in terms of a FSML.

To illustrate, Figure 3.2 shows a feature model of the Struts FSML. The hierarchical nature of the feature model is shown using indentation (subfeatures are further right). Feature cardinality constraints are specified in square brackets and mapping definitions are specified in angle brackets after the name of a feature. For example, the code pattern that the feature `ActionImpl` (Figure 3.2, line 31) corresponds to is a Java class and the code pattern that the feature `forwards` (Figure 3.2, line 40) corresponds to is the method `findForward` in the control flow of that class. The mapping definition of the feature `forwards` uses the mapping type c callsTo: receiver:r [statement:s] and assigns r to the value ActionMapping and s to the value findForward(String). The parameter c is determined implicitly using the context mechanism, which retrieves the value of the parameter from the instance of the closest parent feature with the required mapping type. In this case, the required mapping type is a class since c is a class. The closest such parent feature is the feature `ActionImpl` on line 31 of Figure 3.2 and so, c takes on the value of `ActionImpl`. Any number of instances of the feature `forwards` is possible in a legal

```
[1..1] StrutsApplication <project>
 ![1..1] name (String) <projectName>
  [1..1] StrutsConfig <xmlDocument: '/WEB−INF/struts−config.xml'> <xmlElement name: 'struts−config'>
    [0..*] FormDecl <xmlElements: 'form−beans/form−bean'> <xmlElement>
      [1..1] name (String) <xmlAttribute>
      [1..1] formType (String) <xmlAttribute: 'type'>
      [0..1] isDynaActionForm <valueEqualsTo attribute: ../formType value: 'DynaActionForm'>
      [0..*] formProperty <xmlElements: 'form−property'> <xmlElement>
        [1..1] name (String) <xmlAttribute>
        [0..1] type (String) <xmlAttribute>
    [0..*] ForwardDecl <xmlElements: 'global−forwards/forward'> <xmlElement>
      [1..1] name (String) <xmlAttribute>
      [1..1] path (String) <xmlAttribute>
      [1..1] target (ActionDecl) <where attribute: path equalsTo: ../path>
    [0..*] ActionDecl <xmlElements: 'action−mappings/action'> <xmlElement>
      [1..1] path (String) <xmlAttribute>
      [0..1] name (String) <xmlAttribute>
      [0..1] type (String) <xmlAttribute>
      [1..1] actionImpl (ActionImpl) <where attribute: qualifiedName equalsTo: ../type>
      [0..*] forwards <xmlElements: 'forward'> <xmlElement>
        [1..1] name (String) <xmlAttribute>
        [1..1] path (String) <xmlAttribute>
      [0..1] input (String) <xmlAttribute>
  [0..*] FormImpl <class>
   ![1..1] name (String) <className>
    [0..1] package (String) <qualifier>
    [1..1] qualifiedName (String)
    [0..1] local <isLocal>
   ![1..1] extendsActionForm <assignableTo: 'ActionForm'> <subsumedBy: extendsDynaActionForm>
      [0..1] extendsDynaActionForm <assignableTo: 'DynaActionForm'>
  [0..*] ActionImpl <class>
   ![1..1] name (String) <className>
    [0..1] package (String) <qualifier>
    [1..1] qualifiedName (String)
    [0..1] local <isLocal>
   ![1..1] extendsAction <assignableTo: 'Action'> <subsumedBy: extendsDispatchAction>
      [0..1] extendsDispatchAction <assignableTo: 'DispatchAction'>
        [0..*] actionMethod (String) <methods: 'ActionForward *(ActionMapping, ActionForm,
             HttpServletRequest, HttpServletResponse)'>
      [0..1] overridesExecute <methods: 'ActionForward execute(ActionMapping, ActionForm,
           HttpServletRequest, HttpServletResponse)'>
    [0..*] forwards <callsTo: 'ActionForward ActionMapping.findForward(String)' location: 'ActionForward
         execute(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)' position: 'after'>
      [1..1] name (String) <valueOfArg: 1>
      [1..1] forward <constraint>
        <1−2>
          [0..1] localForward (ForwardDecl) <where attribute: name equalsTo: ../../name> <and attribute: ../
               type equalsTo: ../../../qualifiedName>
          [0..1] globalForward (ForwardDecl) <where attribute: name equalsTo: ../../name> <andParentIs
               instanceOf: 'StrutsConfig'>
    [0..*] inputForwards <callsTo: 'ActionForward ActionMapping.getInputForward()' location: 'ActionForward
         execute(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)' position: 'after'>
      [1..1] name (String) <valueOf attribute: input class: 'ActionDecl'> <where attribute: type equalsTo: ../../
           qualifiedName>
```

Figure 3.2: Struts FSML

configuration as indicated by the cardinality `[0..*]`. The exclamation mark (`!`) before the feature `extendsAction` indicates that the feature for subclassing the Action class is an essential feature. Without the essential feature, the feature's parent won't exist as a concept instance in the framework-specific model. For example, a class `X` is not an Action class if it does not subclass the Action class. Although class `X` satisfies the mapping definition of ActionImpl (Figure 3.2, line 31) as a class, if it does not have the essential feature `extendsAction`, the instance of the feature `ActionImpl` will not be present in the framework-specific model. An example of a feature that corresponds to an XML element is `ForwardDecl` (Figure 3.2, line 11). Its mapping definition specifies that each instance of the feature in a feature configuration corresponds to an XML element on the path `global-forwards/forward`, in the XML document that an instance of its parent feature, `StrutsConfig`, corresponds to. An example of a referential constraint is the two constraints on line 44 of Figure 3.2 of the Struts FSML. The first constraint, <where attribute: name equalsTo: ../../name>, states that the subfeature `name` (Figure 3.2, line 12) of the current feature, which is a `ForwardDecl` feature (Figure 3.2, line 11) with the mapping definition <xmlElements: 'global-forwards/forward'> to refer to the the XML element global-forwards/forward in the XML file struts-config.xml, must equal to the feature `name` (Figure 3.2, line 32) two levels up the tree of the current feature, which is the name of the current Java Action class. Similarly, the second constraint, <and attribute: ../ type equalsTo: ../../../qualifiedName>, states that the feature `type` (Figure 3.2, line 18) in the XML file must equal to the qualified name of the Java Action class (Figure 3.2, line 34). The feature `localForward` and `globalForward` is a feature group and its group cardinality on line 43 of Figure 3.2 states that a forward can be a localForward and/or a globalForward.

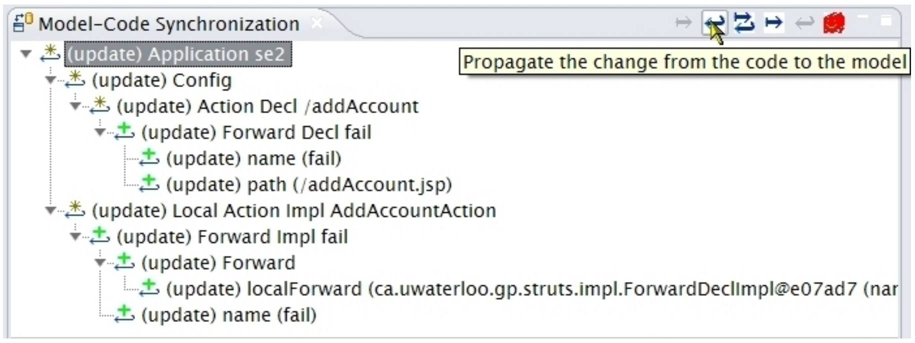
3.2 Round-trip Engineering with Framework-Specific Models

FSMLs are designed to support round-trip engineering, which involves reverse engineering, forward engineering, and synchronization. This section briefly reviews each of these activities.

To reverse engineer a framework-specific model from application code, code queries are run. Code queries use static code analysis to extract instances of framework concepts in a FSML from the application code and have been shown to achieve high precision and recall [Antkiewicz et al., 2008]. The code query for each mapping type from [Antkiewicz, 2008] is reproduced in Appendix B. Prior to invoking the framework-based code assistant described in the next chapter, a framework-specific model must already exists in the project and the model must be extracted from the project code using reverse engineering.

To forward engineer is to generate code from a framework-specific model. Code transformations are executed for each instance of a feature in a framework-specific model. The code transformation for each mapping type from [Antkiewicz, 2008] is reproduced in Appendix C. Forward engineering requires special parameters in the mapping definition for some features that are not necessary in reverse engineering, since code transformation may require additional information that has been abstracted away by code queries. For example, the parameter `location` for the feature `forwards` on line 40 of Figure 3.2 of the Struts FSML takes on the value of `execute(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)` and the parameter `position` takes on the value `after`. The two parameters state that code generated using code transformation should be inserted in the method with the signature `execute(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)` and should be inserted at the end of the method. These two parameters in the FSML set the default location and position of the generated code. The complete list of parameters currently used in forward engineering with FSMLs, from [Antkiewicz, 2008], are reproduced in Table A.3 in Appendix A. For features such as `forwards` where the generated code intertwines with the application logic (Figure 1.1), it is clear that code customization is required after forward engineering.

Round-trip engineering allows models and code to be modified independently and later synchronized through incremental updates. The ability to synchronize changes prevents the target model from being replaced completely in reverse engineering, or the target code from being replaced completely in forward engineering. Prior to the work presented in this thesis, one can edit the model directly in the model editor or edit the code directly, but without any model support. In either case, the user needs to invoke synchronization to keep both the model and code consistent. The user can decide for each change whether to update the change from the model to code, update the change from the code to the model, discard the change, or ignore the change. Figure 3.3 shows a screenshot after manually coding an instance of the two features `forward` (Figure 3.2, line 20 and line 40) in the code editor in a Java class called AddAccountImpl and in the Struts configuration XML file and manually invoking model-code synchronization.


Model–Code Synchronization
Propagate the change from the code to the model

```
(update) Application se2
  (update) Config
    (update) Action Decl /addAccount
      (update) Forward Decl fail
        (update) name (fail)
        (update) path (/addAccount.jsp)
  (update) Local Action Impl AddAccountAction
    (update) Forward Impl fail
      (update) Forward
        (update) localForward (ca.uwaterloo.gp.struts.impl.ForwardDeclImpl@e07ad7 (nar
      (update) name (fail)
```

Figure 3.3: An example of model-code synchronization

Chapter 4

Framework-Based Code Assistant

This chapter describes our framework-based code assistant that is guided by framework-specific models. The code assistant provides context-dependent suggestions and aids the user in the implementation of framework concepts with the consideration of framework constraints. We first give a general methodology for the approach. Then, we discuss a prototype implementation of the framework-based code assistant. Finally, we walk through two examples of using the framework-based code assistant with the prototype, one for the Apache Struts framework and one for the Java applet framework.

4.1 Methodology

This section describes the general methodology for creating a framework-based code assistant. We first state the prerequisites of using the framework-based code assistant. Next, we discuss the criteria for identifying a proposal in the framework-based code assistant. Finally, we present the steps of executing a code assistant proposal.

4.1.1 Prerequisites for Using the Framework-Based Code Assistant

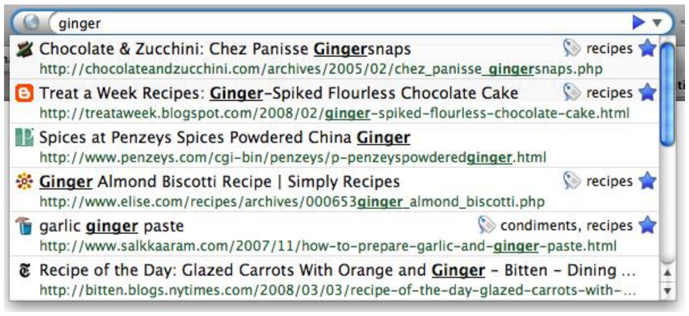
Before invoking a code assistant in the code editor, a framework-specific model of the current working project must already exist in the project, and the model must have been extracted from the project code using reverse engineering. The framework-specific model must contain a feature instance corresponding to the currently opened file in the code editor so that a code context for the code assistant can be retrieved. Since the framework-based code assistant proposes subfeatures of an instantiated feature, it only works for subfeatures (not necessarily direct) of a feature corresponding to a source artifact, such as a Java class file or a XML file.

Also, by the definition of an essential feature, a feature instance corresponding to the opened file may only exist in a framework-specific model if all of its essential features are present and instantiated in the model.

4.1.2 Identifying Code Assistant Proposals

When the code assistant is invoked at a specific location where the code to be inserted is desired, it identifies instances of features whose code pattern contain the cursor position. This process makes use of traceability links from feature instances in the framework-specific model to code, created during reverse engineering as each feature instance is created in the model. Each traceability link maps a feature instance in the framework-specific model to the corresponding part of the code, according to the mapping definition of the FSML feature instantiated. To map the code where the user invokes the code assistant back to the model, the traceability links are interpreted in the reverse direction. However, there might be multiple feature instances corresponding to a single cursor position: when a feature is the context feature of its children features, the code area that the feature covers might be a superset of the code area that its children features cover. For example, the feature `forwards` (Figure 3.2, line 40) in the Struts FSML covers only a method call in the code, whereas its parent feature `ActionImpl` (Figure 3.2, line 31) covers an entire class, including any method call that corresponds to the feature `forwards`. For this reason, we sort the list of code patterns based on the size of the code ranges in increasing order. The rationale is that a feature instance that maps to a smaller code range is more specific and is more likely to be the current context for the user.

Given the feature instances representing the current code context, in the case of suggestion-based autocompletion, the framework-based code assistant lists immediate subfeatures of each feature instance as proposals, with the subfeatures of a feature instance that maps to a smaller code range higher up in the proposal list. However, if adding a subfeature to the model will violate the subfeature's cardinality constraint, the framework-based code assistant filters out the subfeature from the list of proposals. In the case of completion-based autocompletion, all subfeatures that do not contain all of the keywords entered by the user in the feature's name are also filtered out. The keyword support in the framework-based code assistant is comparable to the new "Awesomebar" introduced in a recent version of the internet browser Mozilla Firefox. The Awesomebar has keyword support to autocompletion in the location bar to reduce the need for users to remember prefixes of addresses such that keywords to web page titles and tags can also be matched. Figure 4.1 shows an example of Awesomebar, directly from Mozilla. Also, keyword filtering is more suitable than prefix filtering in framework-based code assistant since framework concepts can be implemented using different mechanisms and the use of keywords can abstract away these different types of mechanisms. The first keyword that the user entered in the set of keywords must also match the name of the FSML the framework-specific model is expressed in. This criterion is especially



```
ginger
Chocolate & Zucchini: Chez Panisse Gingersnaps                    recipes
http://chocolateandzucchini.com/archives/2005/02/chez_panisse_gingersnaps.php
Treat a Week Recipes: Ginger-Spiked Flourless Chocolate Cake      recipes
http://treataweek.blogspot.com/2008/02/ginger-spiked-flourless-chocolate-cake.html
Spices at Penzeys Spices Powdered China Ginger
http://www.penzeys.com/cgi-bin/penzeys/p-penzeyspowderedginger.html
Ginger Almond Biscotti Recipe | Simply Recipes                    recipes
http://www.elise.com/recipes/archives/000653ginger_almond_biscotti.php
garlic ginger paste                                   condiments, recipes
http://www.salkkaaram.com/2007/11/how-to-prepare-garlic-and-ginger-paste.html
Recipe of the Day: Glazed Carrots With Orange and Ginger - Bitten - Dining ...
http://bitten.blogs.nytimes.com/2008/03/03/recipe-of-the-day-glazed-carrots-with-...
```

Figure 4.1: Awesomebar in Mozilla Firefox

important in the case when multiple framework-specific models exist in the project, representing different frameworks used in the project's application code.

4.1.3 Executing a Code Asssistant Proposal

When the user selects a proposal from the list of proposals in the framework-based code assistant, the code assistant first opens a new transaction. It then instantiates the feature corresponding to the proposal and all its mandatory subfeatures in the model in a depth-first manner. The feature instance instantiated for the feature corresponding to the proposal uses the location information in the current cursor position so that the generated code for the feature instance later produced using code transformation can be placed in the current cursor position. If a feature to be instantiated is a feature group, the different alternatives in the feature group are presented to the user. After the user selects a feature from the feature group so that the group constraint is satisfied, the instantiation of other features continues. If a feature to be instantiated requires values in the mapping definition that are missing, the user is prompted to provide the missing values. For example, the feature `name` (Figure 3.2, line 32) does not have a default value and its instantiation requires the user to input a value corresponding to the name of the Action class. If a feature, such as the feature on line 42 of Figure 3.2, has the mapping definition "constraint", it means that the subfeatures of the feature contain a referential constraint on another feature in the current level and so, the code assistant processes the feature with the mapping definition "constraint" before other features on the same level in the feature model. If a feature has a referential integrity constraint between two features as its mapping definition, one of which requires user inputs, the code assistant make suggestions for the user input based on the other feature in the constraint. Alternatively, the user can enter a new value for the feature that requires an input value, in which case, the other feature in the constraint is also instantiated, keeping the referential integrity constraint satisfied. For example, the constraint on line 45 of Figure 3.2 allows the list of global forwards declared in the XML file to be presented to the user as suggestions to the input for the feature `name` (Figure 3.2, line 41). If the user chooses to enter a new value for the feature `name`, the code assistant instantiates a new instance of the feature `name` (Figure 3.2, line

12) under `ForwardDecl` so that the referential integrity constraint is honoured. As each feature is instantiated, the feature's associated code transformation is stored in a queue. After all features are instantiated for the proposal in the model, the code assistant executes the queue of code transformation in order and the generated code is highlighted in the code editor. If all feature instantiation and code transformation succeeds, the changed model and the changed code are committed to the file system, and the code assistant closes the transaction.

4.2 Implementation

To demonstrate the feasibility of the approach, we implemented the framework-based code assistant described in the previous section as a set of Eclipse plugins. The current implementation provides framework-based code assistance for both Java and XML files, each extending the content assist mechanism in Eclipse. The framework-based code assistant plugin for Java implements the extension point `org.eclipse.jdt.ui.javaCompletionProposalComputer` in Eclipse, which contributes to the proposal computer for the content assist process in the default Java editor. For XML, as such an extension point does not currently exist in Eclipse for the default XML editor, we wrote a new XML editor with framework-based code assistant support. Both the Java framework-based code assistant plugin and the XML framework-based code assistant plugin works for any FSMLs. In other words, the number of Eclipse plugins corresponds to the number of types of source artifacts that needs to be supported, but is independent on the number of frameworks that needs to be supported. A new framework can be supported as long as a FSML is written for the framework, and all the required code queries and code transformations, which are reusable among different frameworks, have been implemented to support reverse engineering and forward engineering.

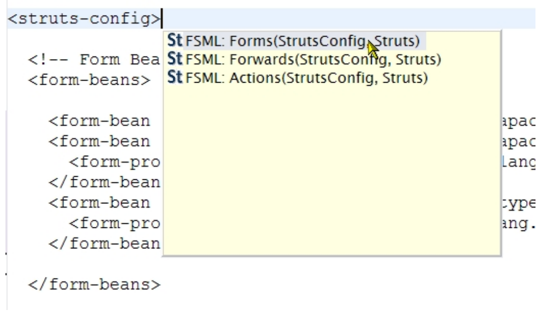
When the user presses Ctrl-Space, the shortcut key for content assist in Eclipse, in either the XML or the Java code editor, the code assistant searches in the project for all framework-specific models expressed using any known FSMLs. The code assistant then presents a list of proposals to the user based on interpreting the models described in section 4.1.2 and sorts the proposals by the size of the corresponding context feature instance's code pattern's size range. The syntax for each proposal name is:

```
FSML: feature name, (context name, framework name)
```

Once the user selects a proposal, whenever code transformations in forward engineering requires a value for a parameter in the mapping definition from the user, the framework-based code assistant displays a popup dialog box to the user, prompting the user for the value. The code assistant also presents suggestions through constraints in a drop-down menu in a dialog box.

The existing infrastructure for round-trip engineering handles the forward engineering for each feature. Forward engineering for Java uses the Eclipse Java Development Tools (JDT) API, whereas forward engineering for XML uses the Eclipse Web Tools Platform (WTP) API. Both APIs manipulate the abstract representation of the file, which is abstract syntax tree (AST) for Java or document object model (DOM) for XML, and generates code from the abstract representation before applying pretty-printing onto the generated code.

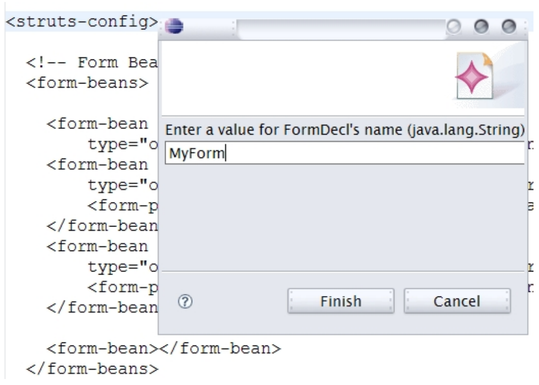
Figure 4.2 shows a few screenshots of a simple example of framework-based code assistant for the XML editor in action. The next two sections describe in detail two more sophisticated examples of the framework-based code assistant invoked from the Java editor for two different applications, each using a different framework.



```
<struts-config>

  <!-- Form Bea
  <form-beans>

    <form-bean                                        apac
    <form-bean                                        apac
      <form-pro                                       lang
    </form-bean
    <form-bean                                        type
      <form-pro                                       ang.
    </form-bean

  </form-beans>
```

St FSML: Forms(StrutsConfig, Struts)
St FSML: Forwards(StrutsConfig, Struts)
St FSML: Actions(StrutsConfig, Struts)

(a) Proposals in XML Editor

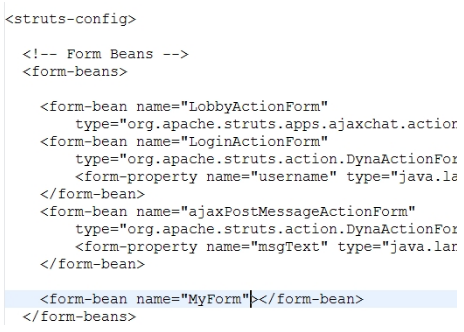


```
<struts-config>

  <!-- Form Bea
  <form-beans>

    <form-bean
        type="o
    <form-bean
        type="o
        <form-p
    </form-bean
    <form-bean
        type="o
        <form-p
    </form-bean

    <form-bean></form-bean>
  </form-beans>
```

Enter a value for FormDecl's name (java.lang.String)

MyForm

(b) Entering the value for `FormDecl`'s feature `name`



```
<struts-config>

  <!-- Form Beans -->
  <form-beans>

    <form-bean name="LobbyActionForm"
        type="org.apache.struts.apps.ajaxchat.action
    <form-bean name="LoginActionForm"
        type="org.apache.struts.action.DynaActionFor
        <form-property name="username" type="java.la
    </form-bean>
    <form-bean name="ajaxPostMessageActionForm"
        type="org.apache.struts.action.DynaActionFor
        <form-property name="msgText" type="java.lan
    </form-bean>

    <form-bean name="MyForm"></form-bean>
  </form-beans>
```

(c) Resulting XML Code

Figure 4.2: Framework-based code assistant in XML editor

4.3 An Example Walk-through for an Apache Struts Application

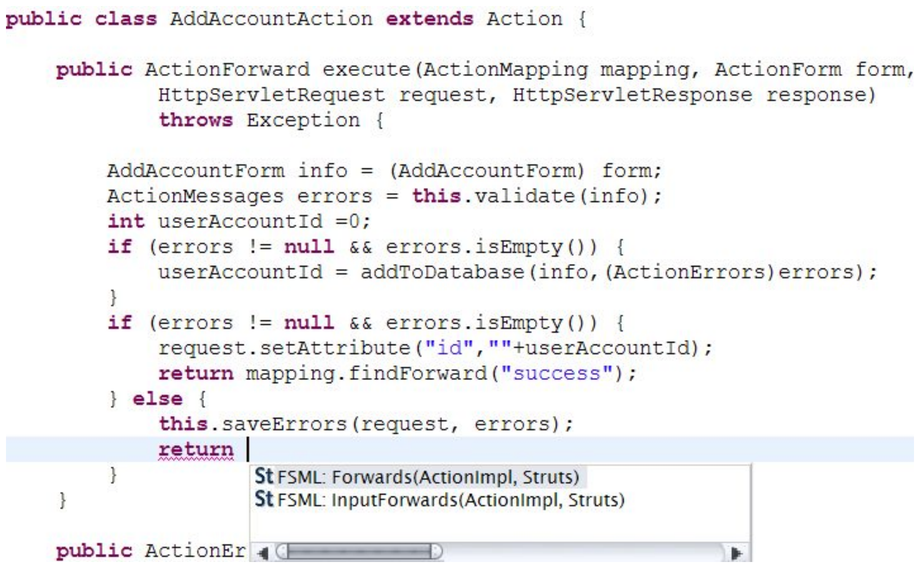
This section illustrates the use of the prototype implementation of framework-based code assistant by walking through an example of its use in an Apache Struts application.

Suppose the user is in a Struts Action class called AddAccountAction in the Java editor. In addition to forwarding to the page "success", defined in the existing struts-config.xml, the user wishes to forward to the page "failure" if an error occurs. Figure 4.3 shows a textual rendering of the original framework-specific model in the project.

```
 1 StrutsApplication <project>
 2   StrutsConfig
 3     ActionDecl
 4       path ('addAccount')
 5       name ('AddAccountForm')
 6       type ('action.AddAccountAction')
 7       actionImpl (ActionImpl AddAccountAction)
 8       ForwardDecl
 9         name ('success') <xmlAttribute>
10         path ('/addAccount2.jsp') <xmlAttribute>
11   ActionImpl
12     name ('AddAccountAction')
13     qualifiedName ('action.AddAccountAction')
14     extendsAction
15     forwards
16       name ('success')
17       forward
18         localForward (ForwardDecl success)
```

Figure 4.3: Original framework-specific model for Apache Struts application example

When the user presses Ctrl-Space to invoke the framework-based code assistant in the Java editor, because the Action class instantiates the concept `ActionImpl` (Figure 4.3, line 11), the reverse navigation will first identify that concept's instance in the framework-specific model. Next, by interpreting the metamodel of the framework-specific model, the Struts FSML, the code assistant will propose the features that can be instantiated in the feature instance, `forwards` (Figure 3.2, line 40) and `inputForwards` (Figure 3.2, line 46), as shown in Figure 4.4. After the user selects the proposal `forwards`, the framework-based code assistant creates an instance of the feature `forwards` in the framework-specific model and stores its associated code transformation, the method call `mapping.findForward(null)` at the current cursor position, in a queue. Notice that, because the feature `forwards` specifies that the receiver of the method call is of type ActionMapping, the framework-based code assistant automatically uses the only variable of type ActionMapping in the current scope to generate the method call. Next, the code assistant proceeds to process all the subfeatures of the feature `forwards` that have the mapping definition "constraint" and sees that the feature `forward` (Figure 3.2, line 42) is such a feature and

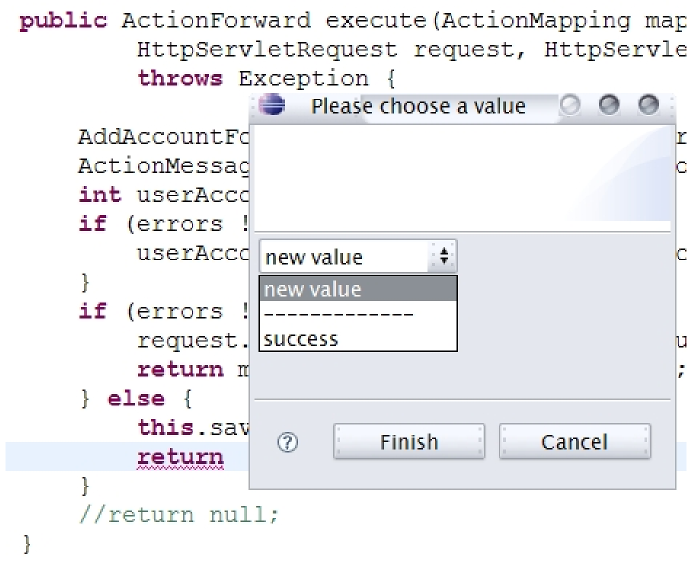


```
public class AddAccountAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {

        AddAccountForm info = (AddAccountForm) form;
        ActionMessages errors = this.validate(info);
        int userAccountId =0;
        if (errors != null && errors.isEmpty()) {
            userAccountId = addToDatabase(info,(ActionErrors)errors);
        }
        if (errors != null && errors.isEmpty()) {
            request.setAttribute("id",""+userAccountId);
            return mapping.findForward("success");
        } else {
            this.saveErrors(request, errors);
            return |
        }
    }

    public ActionEr
```

St FSML: Forwards(ActionImpl, Struts)
St FSML: InputForwards(ActionImpl, Struts)

Figure 4.4: Framework-based code assistant for Struts

is mandatory. It asks the user to make a choice between the feature groups since at least one and at most two features in the group are required. Should the user chooses `localForward` instead of `globalForward`, the code assistant instantiates the feature and proceeds to process the feature `localForward` (Figure 3.2, line 44). The feature `localForward` contains a referential integrity constraint that specifies that the value of the feature `name` (Figure 3.2, line 43) must match the value of the feature `name` (Figure 3.2, line 12), and the value of the feature `type` (Figure 3.2, line 18) in the forward declaration must match the value of the feature `qualifiedName` (Figure 3.2, line 34) in the Action class. The framework-based code assistant thus displays a dialog box with suggestions to the user for the possible values of `name` (Figure 3.2, line 41). The dialog box suggests the value "success" and provides an option for the user to enter a new value, as in Figure 4.5. Since the user wants "failure" instead of "success", the user chooses the new value and enters the value "failure". At this time, the code assistant automatically creates a new instance of the feature `ForwardDecl` (Figure 3.2, line 11) and sets the value of the subfeature `name` (Figure 3.2, line 12) and the value of the feature `name` (Figure 3.2, line 41) to "failure". The code assistant also stores in a queue the associated code transformation for the feature `name` in `ForwardDecl`, which is `<forward name="failure">` under the AddAccountAction XML declaration in Struts-Config.xml. Also, the user will be prompted for a value for the feature `path` (Figure 3.2, line 13) since it is mandatory. After the user enters "`/addAccount.jsp`", the feature is instantiated and its associated code transformation, an XML attribute for the XML declaration for `failure`, is also stored in the queue. After the feature `localForward` is processed, the code assistant returns to process the feature `name`(Figure 3.2, line 41) that it has deferred before to first process the feature `forward` with the mapping definition "constraint". The feature already has the value "failure" from before, so its associated code transformation, a replacement of the method call `findForward(null)`



```
public ActionForward execute(ActionMapping map
        HttpServletRequest request, HttpServle
        throws Exception {

    AddAccountFo                                r
    ActionMessag                                c
    int userAcco
    if (errors !
        userAcco                                c
    }
    if (errors !
        request.                                u
        return m                                ;
    } else {
        this.sav
        return
    }
    //return null;
}
```

Figure 4.5: Suggestions for the feature `forwards` in Apache Struts

argument to the value “failure”, is stored in a queue. Since no more mandatory subfeatures exist, the queue of code transformations is executed and the changes to the framework-specific model are committed. As a result of executing a proposal from the code assistant, the code below are added to the Struts configuration XML file and a method call `mapping.findForward(‘‘failure’’)` has been generated at the cursor position.

```
<action path="/addAccount" type="action.AddAccountAction"
                name="AddAccountForm">
        <forward name="success" path="/addAccount2.jsp"/>
        <forward name="failure" path="/addAccount.jsp"/>
</action>
```

Figure 4.6 shows the resulting framework-specific model.

```
StrutsApplication <project>
  StrutsConfig
    ActionDecl
      path ('addAccount')
      name ('AddAccountForm')
      type ('action.AddAccountAction')
      actionImpl (ActionImpl AddAccountAction)
      ForwardDecl
        name ('success') <xmlAttribute>
        path ('/addAccount2.jsp') <xmlAttribute>
      ForwardDecl
        name ('failure') <xmlAttribute>
        path ('/addAccount.jsp') <xmlAttribute>
  ActionImpl
    name ('AddAccountAction')
    qualifiedName ('action.AddAccountAction')
    extendsAction
    forwards
      name ('success')
      forward
        localForward (ForwardDecl success)
    forwards
      name ('failure')
      forward
        localForward (ForwardDecl failure)
```

Figure 4.6: Resulting framework-specific model for Apache Struts application example

4.4 Another Example-Walk Through for a Java Applet

This section walks through another example of using the framework-based code assistant on an application implemented with a different framework, the Java applet framework. Figure 4.7 shows a feature model of the Applet FSML.

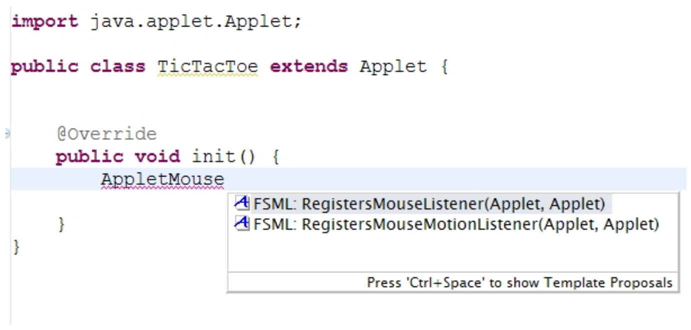
```
AppletModel <project>
  [0..*] Applet <class>
    [1..1] name (String) <fullyQualifiedName>
   ![1..1] extendsApplet <assignableTo: 'Applet' local: true> <subsumedBy: extendsJApplet>
      [0..1] extendsJApplet <assignableTo: 'JApplet'>
    [0..1] overridesLifecycleMethods
     !<1-5>
        [0..1] init <methods: 'void init()'>
        [0..1] start <methods: 'void start()'>
        [0..1] paint <methods: 'void paint(Graphics)'>
        [0..1] stop <methods: 'void stop()'>
        [0..1] destroy <methods: 'void destroy()'>
    [0..*] showsStatus <callsReceived: 'void Applet.showStatus(String)' location: 'void init()' position: 'after'>
      [0..*] message (String) <valueOfArg: 1>
    [0..*] registersMouseListener <callsTo: 'void Component.addMouseListener(MouseListener)' position: 'after'
          location: 'void init()'>
     !<1-1>
        [0..1] this <argumentIsThis: 1>
          [1..1] implementsMouseListener <assignableTo: 'MouseListener'>
          [1..1] deregisters <callsTo: 'void Component.removeMouseListener(MouseListener)' location: 'void
               destroy()'>
           ![1..1] this <argumentIsThis: 1>
        [0..1] mouseListenerField <argumentIsField: 1> <field>
          [1..1] listenerField (String) <fieldName>
          [1..1] typedMouseListener <fieldOfType: 'MouseListener'>
      [1..1] initialized <assignedNew: ‘void MouseListener()’ initializer: true>
      [1..1] deregisters <callsTo: ‘void Component.removeMouseListener(MouseListener)’ location: ‘void destroy()’>
       ![1..1] field <argumentIsField: 1 sameAs: ../../listenerField>
[0..*] registersMouseMotionListener <callsTo: ‘void Component.addMouseMotionListener(MouseMotionListener)’ position: ‘after’ location: ‘void init()’>
 !<1−1>
    [0..1] this <argumentIsThis: 1>
      [1..1] implementsMouseMotionListener <assignableTo: ‘MouseMotionListener’>
      [1..1] deregisters <callsTo: ‘void Component.removeMouseMotionListener(MouseMotionListener)’ location: ‘void destroy()’>
       ![1..1] this <argumentIsThis: 1>
    [0..1] mouseMotionListenerField <argumentIsField: 1> <field>
      [1..1] listenerField (String) <fieldName>
      [1..1] typedMouseMotionListener <fieldOfType: ‘MouseMotionListener’>
      [1..1] initialized <assignedNew: ‘void MouseMotionListener()’ initializer: true>
      [1..1] deregisters <callsTo: ‘void Component.removeMouseMotionListener(MouseMotionListener)’ location: ‘void destroy()’>
       ![1..1] field <argumentIsField: 1 sameAs: ../../listenerField>
[0..*] registersKeyListener <callsTo: ‘void Component.addKeyListener(KeyListener)’ position: ‘after’ location : ‘void init()’>
 !<1−1>
    [0..1] this <argumentIsThis: 1>
      [1..1] implementsKeyListener <assignableTo: ‘KeyListener’>
      [1..1] deregisters <callsTo: ‘void Component.removeKeyListener(KeyListener)’ location: ‘void destroy()’>
       ![1..1] this <argumentIsThis: 1>
    [0..1] keyListenerField <argumentIsField: 1> <field>
      [1..1] listenerField (String) <fieldName>
      [1..1] typedKeyListener <fieldOfType: ‘KeyListener’>
      [1..1] initialized <assignedNew: ‘void KeyListener()’ initializer: true>
      [1..1] deregisters <callsTo: ‘void Component.removeKeyListener(KeyListener)’ location: ‘void destroy()’>
       ![1..1] field <argumentIsField: 1 sameAs: ../../listenerField>
[0..*] Thread <field>
  [1..1] thread (String) <fieldName>
 ![1..1] typedThread <fieldOfType: ‘Thread’>
  [1..1] InitializesThread
   !<1−1>
      [0..1] initializesThreadWithRunnable <assignedNew: ‘void Thread(Runnable)’ position: ’after’ location : ‘void init()’>
        <1−1>
          [0..1] this <argumentIsThis: 1>
            [1..1] implementsRunnable <assignableTo: ‘Runnable’>
          [0..1] helper <argumentIsNew: 1 signature: ‘void Runnable()’>
          [0..1] variable (String) <argumentIsVariable: 1 signature: ‘void Runnable()’>
          [0..1] runnableField <argumentIsField: 1> <field>
            [1..1] typedRunnable <fieldOfType: ‘Runnable’>
            [1..1] name (String) <fieldName>
            [1..1] initialized <assignedNew: ‘void Runnable()’ initializer: true>
      [0..1] initializesWithThreadSubclass <assignedNew initializer: true subtypeOf: ‘Thread’> <class>
        [1..1] name (String) <fieldType> <fullyQualifiedName>
        [1..1] overridesRun <methods: ‘void run()’>
        [1..1] extendsThread <assignableTo: ‘Thread’>
  [1..1] nullifiesThread <assignedNull location: ‘void destroy()’ position: ‘after’>
[0..*] singleTaskThread <callsTo: ‘void Thread(Runnable)’ position: ’after’ location: ‘void init()’ statement: true>
  <1−1>
    [0..1] runnable <argumentIsNew: 1 signature: ‘void Runnable()’>
    [0..1] runnableField <argumentIsField: 1> <field>
      [1..1] typedRunnable <fieldOfType: ‘Runnable’>
      [1..1] name (String) <fieldName>
      [1..1] initialized <assignedNew: ‘void Runnable()’ initializer: true>
[0..*] parameter <callsReceived: ‘String Applet.getParameter(String)’ location: ‘void init()’>
  [0..*] name <valueOfArg: 1>
[0..1] providesParameterInfo <methods: ‘String[][] getParameterInfo()’>
[1..1] providesInfoForParameters <constraint: ../parameter implies: ../providesParameterInfo>
```

Figure 4.7: Applet FSML

Suppose the user is writing a Java applet for the game of Tic-tac-toe and wishes to first implement the part where the Java applet will respond to mouse clicks. The framework-specific model of the user's current code is shown in Figure 4.8. The user types in the keyword "Applet" and "Mouse", as in Figure 4.9, and the framework-based code assistant proposes the two features containing the word "Mouse" that can be instantiated in the current feature instance `Applet` (Figure 4.8, line 1): registersMouseListener (Figure 4.7, line 15) and registersMouseMotionListener (Figure 4.7, line 27).

```
1  Applet
2    name ('TicTacToe')
3    extendsApplet
4    overridesLifecyleMethods
5      init
```

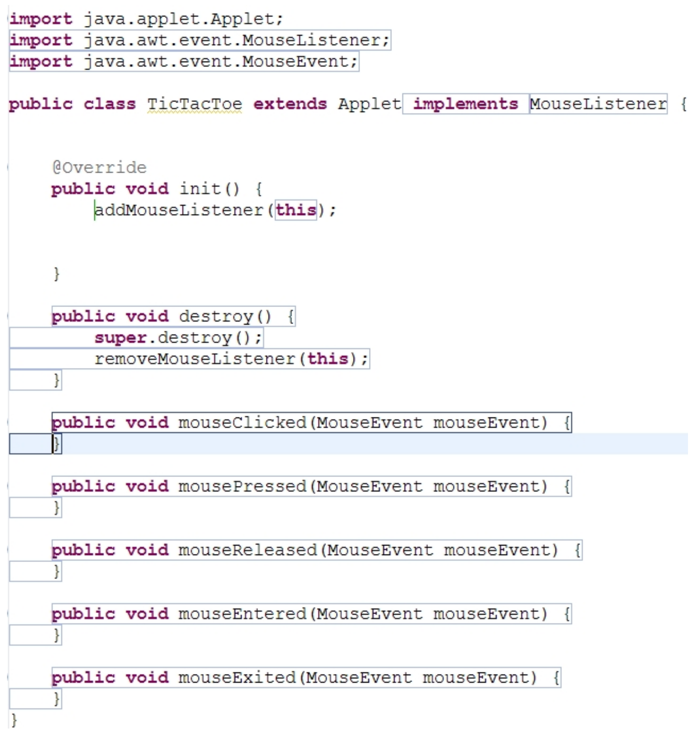
Figure 4.8: Original framework-specific model for Java applet example



```
import java.applet.Applet;

public class TicTacToe extends Applet {


    @Override
    public void init() {
        AppletMouse
                        FSML: RegistersMouseListener(Applet, Applet)
    }                   FSML: RegistersMouseMotionListener(Applet, Applet)
}
                                   Press 'Ctrl+Space' to show Template Proposals
```

Figure 4.9: Keyword programming for Java applet example

The user selects the `RegistersMouseListener` proposal and the framework-based code assistant creates an instance of the `registerMouseListener` feature. Its associated code transformation, an `addMouseListener(null)` call at the current cursor position, is also stored in a queue. Since the feature also contains a feature group that is marked essential, the code assistant proposes the two features in the group, `this` and `mouseListenerField`, to the user. Suppose the user chooses `this`, the code assistant creates an instance of the feature `this`, together with storing its associated code transformation in a queue, and the code assistant continues the process with its subfeatures. The two features, `implementsMouseListener` (Figure 4.7, line 18) and `deregisters` (Figure 4.7, line 19), are all then instantiated with its code transformation in the queue, before the subfeature of `deregisters`, `this` (Figure 4.7, line 20), which refers to the argument of the deregister call, and its associated code transformation. The resulting code is shown in Figure 4.10, and the resulting framework-specific model is shown in Figure 4.11.



```java
import java.applet.Applet;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;

public class TicTacToe extends Applet implements MouseListener {


    @Override
    public void init() {
        addMouseListener(this);


    }

    public void destroy() {
        super.destroy();
        removeMouseListener(this);
    }

    public void mouseClicked(MouseEvent mouseEvent) {
    }

    public void mousePressed(MouseEvent mouseEvent) {
    }

    public void mouseReleased(MouseEvent mouseEvent) {
    }

    public void mouseEntered(MouseEvent mouseEvent) {
    }

    public void mouseExited(MouseEvent mouseEvent) {
    }
}
```

Figure 4.10: Resulting code for Java applet example

```
Applet
  name ('TicTacToe')
  extendsApplet
  overridesLifecyleMethods
    init
  registerMouseListener
    this
      implementsMouseListener
      deregisters
        this
```

Figure 4.11: Resulting framework-specific model for Java applet example

Chapter 5

Evaluation

This chapter presents a study, conducted in two phases, that evaluates the proposed model-guided framework-based code assistant approach to code generation. The study tests the hypothesis that the framework-based code assistant approach to code generation reduces the effort to produce framework application code when compared to forward engineering from the model editor.

In general, the framework-based code assistant is able to reduce the effort in producing code comparing to forward engineering from the model editor in several ways. Firstly, unlike forward engineering from the model editor, framework-based code assistant does not require on-demand synchronization between the code and the model since as a code assistant proposal executes, the code assistant not only creates new feature instances in the model, but also generates code for the new feature instances, automatically keeping both the model and the code synchronized. Secondly, unlike the current implementation of the model editor that requires new feature instances to be created one-by-one, our framework-based code assistant automatically creates instances of all mandatory subfeatures recursively. Thirdly, framework-based code assistant allows the user to work directly in the code editor instead of having to switch back and forth between the model editor and the code editor, which both reduces the user's cognitive load from context switches and avoids the need to synchronize changes between model and code. Lastly, framework-based code assistant generates code at the location where the user invokes the code assistant, avoiding the need for the user to move the generated code to the desired location after forward engineering from the model editor that would again require switching from the model editor to the code editor. The evaluation in this section concentrates on the last advantage and attempts to further understand the need to move code in the old approach of forward engineering from the model editor, and how much of this effort can be reduced by using the framework-based code assistant instead.

The first phase of the study analyzes every feature in both the Applet FSML and the Struts FSML by reverse engineering a set of sample applications instantiating the two frameworks and classifying the variability in the code pattern location in the feature instances, so that the extent to which the generated code needs to be moved can be assessed in the next phase. The second phase of the study involves calculating the extent that moving the generated code is required using forward engineering from the model editor and compares the result with the framework-based code assistant approach. We then manually inspect the code with varying code pattern location to further refine the result.

5.1 Test Data

The study uses a set of a total of 90 sample applications, 84 of which are Java applets and 6 of which are applications that uses the Apache Struts framework. The set of applications is the same set of applications that was used in [Antkiewicz et al., 2008] to evaluate the precision and recall of code queries for reverse engineering. We grouped all the Java applets into one project and created a project for each of the 6 Apache Struts applications. Applications for both frameworks contain a mix of sample applications shipped with the framework and other open source applications. The complete list of applications is reproduced for convenience in Appendix D.

5.2 Phase 1 of the Study

5.2.1 Setup of Phase 1

In the first phase of the study, we first extracted a framework-specific model for each of the projects and then recorded the number of feature instances for each FSML feature. We then analyzed the mapping definition for each feature in both FSMLs to see whether, without the forward engineering parameters (Table A.3), there are other points of variation in the location of the code patterns that are not covered by the other parameters in the mapping definition. For example, given the mapping definition for the feature `extendsApplet` in the Applet FSML, *c* assignableTo: *Applet* local:true, the corresponding code pattern must be in class *c*, as `class c extends Applet`. On the other hand, given the mapping definition of the feature `showStatus` in the Applet FSML, *c* callsReceived: *void Applet.showStatus(String)* location:*void init()* position:*after*, ignoring the forward engineering parameters location and position, the corresponding code pattern can be in any line within any method body. For each feature that has a varying location, we grouped its feature instances by the method that the code patterns appear in. We also manually post-process the data to ensure that for code that appears in a helper method with no other code inside it, instead of using the helper method as the location, the method the call site resides in is used in the data instead, since neither code assistant nor forward engineering from model editor generates helper methods.

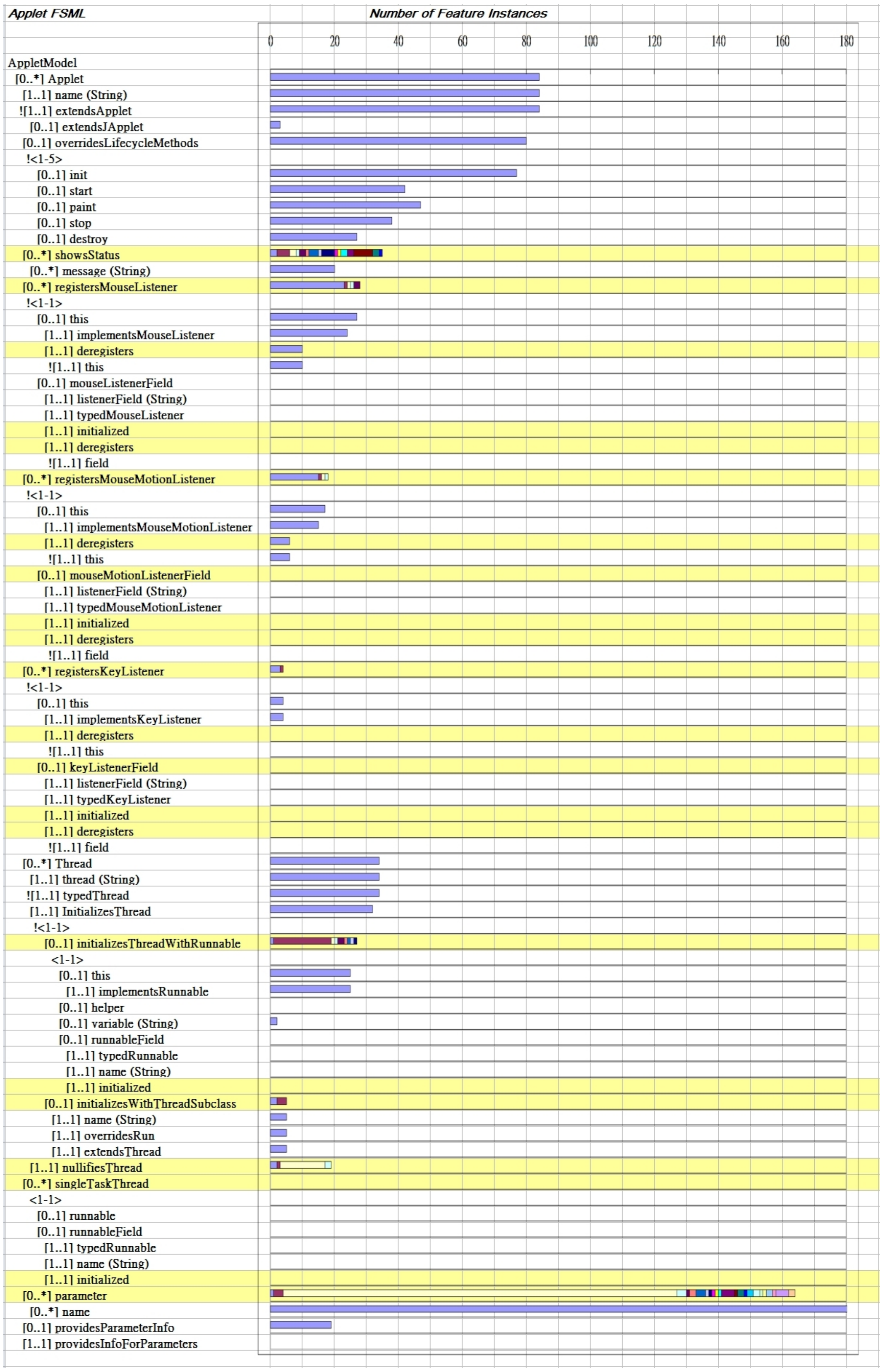


Figure 5.1: Variability in code pattern location for feature instances in sample applications for the Java applet framework

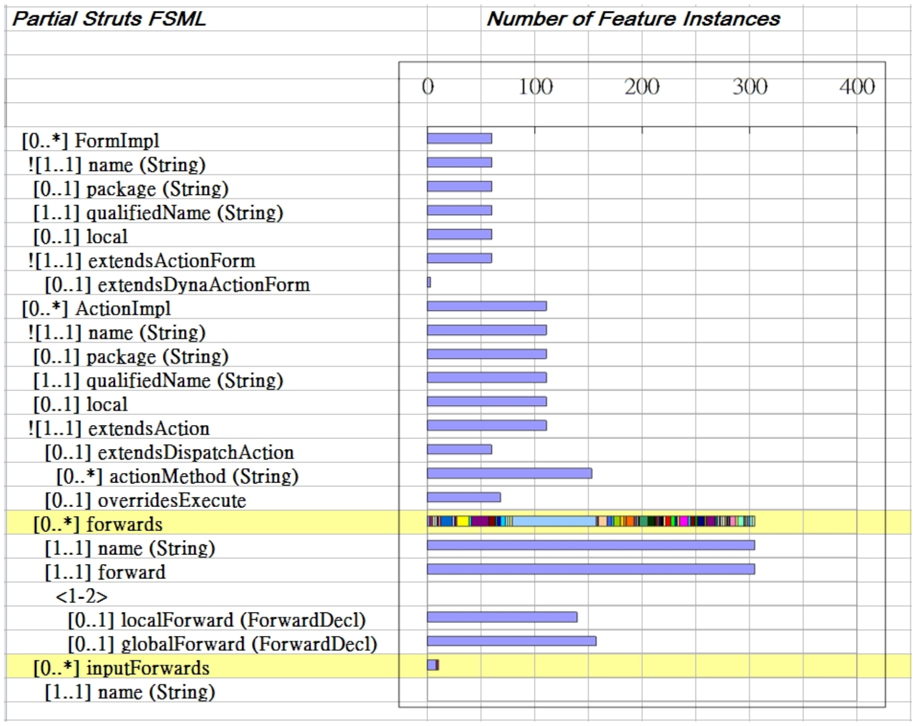


Figure 5.2: Variability in code pattern location for feature instances in sample applications for the Apache Struts framework

5.2.2 Results and Analysis of Phase 1

The result of phase 1 of the study is presented in Figure 5.1, for the entire set of sample applications for the Java applet framework, and in Figure 5.2, for the applications for the Apache Struts framework. The first column of both graphs shows the names of each feature in the the corresponding FSML and the second column records the number of feature instances of each feature found in the sample applications, grouped by the method that the code patterns appear in. The rows for all the features that have varying code pattern location are highlighted.

We first examine the number of feature instances in the set of sample applications. Interestingly, numerous features have no feature instances, but there are a few features with many feature instances. For example, there are a total of 164 instances of the feature `parameter`, averaging almost 2 instances per Java applet. There are also as many as 498 instances of the subfeature `name` for `parameter`, since data flow analysis in reverse engineering matches all possible values of the `name` parameter in the `parameter` method call. Since the subfeature `name` for `parameter` does not have varying location and is not relevant in this study, the full number is cut off and is not shown in the graph due to space limitations.

Next, we examine the features that have varying locations. Out of a total of 71 features in the Applet FSML that have mapping definitions, a total of 22 features, or 31%, have varying location. For the Struts FSML, we omitted the features that map to XML code patterns because they do not vary in location. Only 2 features out of 23 features that have Java mapping definitions in the Struts FSML have varying location. After examining each of these features in both FSMLs carefully, we found that these are features with either the forward engineering parameter `location` or the forward engineering parameter `initializer`. The forward parameter location is as expected, since it specifies the method that the generated code should be inserted into. The forward engineering parameter initializer specifies whether a field assignment should be created in the field's initializer. Normally, the forward engineering parameter initializer is set to true as a default value in mapping definitions. However, if the field assignment is not created in the field's initializer, there are also other locatioins where this pattern can be generated, namely, when the field assignment is in some other method bodies, away from the field's initializer.

For the Applet FSML, we can see that 12 of the 22 features that can have varying location have no feature instances in the sample set of 84 applications. Out of the 10 features that have one or more feature instances, 8 feature instances have varying location, and 2 have no varying location. It is interesting to see that both of the features with no varying location are deregister calls and are mandatory subfeatures of the corresponding register calls. In other words, when the user selects the proposal from the code assistant for the register calls, the deregister calls should be generated automatically. For the 8 features with varying locations, the feature `showStatus` is the only feature that has no clear single best location. This result might not be as surprising if one understands the concept of showStatus in Java applet. `ShowStatus` displays a message in the status window and, depending on the application logic, different messages are usually displayed. For this reason, the code pattern for the `showStatus` feature instances appear inside a large variety of method bodies. The other 7 features all have one location where the code patterns appear in at least more than 20% of the time.

For the Struts FSML, both features with varying locations have a single best location. However, for the feature `forwards`, the best location, which is the `execute` callback method in an Action class, still accounts for only 23% of the total number of feature instances. Further analysis reveals that 83% of the feature instances of the feature `forwards` also instantiates the feature `extendsDispatchAction`. Dispatch Action is a special type of Action class that, instead of dispatching to the default callback method named `execute`, dispatches to a public method that is named by the request parameter whose name is specified in the Struts XML configuration file. For example, with the XML code below in the Struts XML configuration file, the value of the request parameter with the name "method" will be the method that will be dispatched to in the Action class.

```
<action path="/saveSubscription"
        type="org.apache.struts.actions.DispatchAction"
        name="subscriptionForm" scope="request"
        input="/subscription.jsp" parameter="method"/>
```

Therefore, the URL

```
http://localhost:8080/myapp/saveSubscription.do?method=update
```

will execute the method `update` in the saveSubscription Action class. In that case, since the value of the request parameter can only be determined at runtime, only the developer knows the location the code for the concept `forwards` should be in.

Given the analysis above, we can categorize the features into `anchored` and `floatable`. Anchored features are ones that are not possible to have a varying location for their code patterns and are all features whose rows are not highlighted in Figure 5.1. Floatable features are ones that are possible to have a varying location for their code patterns. Floatable features can be further categorized as `non-floating`, `semi-floating`, and `free-floating`. Non-floating features are floatable features that have code patterns that usually occur in a fixed location. Semi-floating are floatable features that have code patterns that have varying locations, but one location dominates more than 20% of the time and appears to be a good default location. Free-floating features are floatable features that have code patterns that have no clear defaults and seem to have unpredictable locations.

5.3 Phase 2 of the Study

5.3.1 Setup of Phase 2

From phase 1 of the study, we calculate the percentage and the number of feature instances that are not handled by the default location. These are feature instances that would require the user to move the generated code from the default location to the desired location when using forward engineering from the model editor. The framework-based code assistant, on the other hand, can safely be invoked in the desired location as the user programs and generates the code in place hence, requiring no further customization. Since phase 1 of the study concentrated on the variability in the location, namely, the method that the code patterns is in, this phase involves inspecting the code manually to see whether the line that the code pattern appears in in a method body also matters, or in other words, whether the code patterns entangles with application logic. Code patterns that entangle with the application logic are code that even with a default location, which is the method to place the code patterns in, requires further customization to integrate the generated code with the application logic inside a method body. In other words, these code patterns that entangle with the application logic code again require code customization by moving the code within a method body when using forward engineering from the model editor, but not when using framework-based code assistant.

Partial Applet FSML	**>0 Instances**	**# of Instances without Defaults**	**% of Instances without Defaults**	**Can Code Entangle with Application Logic?**
[0..*] showsStatus	Yes	35	100.00%	Yes
[0..*] message (String)				
[0..*] registersMouseListener	Yes	5	17.86%	No
!<1-1>				
[0..1] this				
[1..1] implementsMouseListener				
[1..1] deregisters	Yes	0	0.00%	No
![1..1] this				
[0..1] mouseListenerField				
[1..1] listenerField (String)				
[1..1] typedMouseListener				
[1..1] initialized	No	N/A	N/A	N/A
[1..1] deregisters	No	N/A	N/A	N/A
![1..1] field				
[0..*] registersMouseMotionListener	Yes	3	16.67%	No
!<1-1>				
[0..1] this				
[1..1] implementsMouseMotionListener				
[1..1] deregisters	Yes	0	0.00%	No
![1..1] this				
[0..1] mouseMotionListenerField	No	N/A	N/A	N/A
[1..1] listenerField (String)				
[1..1] typedMouseMotionListener				
[1..1] initialized	No	N/A	N/A	N/A
[1..1] deregisters	No	N/A	N/A	N/A
![1..1] field				
[0..*] registersKeyListener	Yes	1	25.00%	No
!<1-1>				
[0..1] this				
[1..1] implementsKeyListener				
[1..1] deregisters	No	N/A	N/A	N/A
![1..1] this				
[0..1] keyListenerField	No	N/A	N/A	N/A
[1..1] listenerField (String)				
[1..1] typedKeyListener				
[1..1] initialized	No	N/A	N/A	N/A
[1..1] deregisters	No	N/A	N/A	N/A
![1..1] field				
[0..*] Thread				
[1..1] thread (String)				
![1..1] typedThread				
[1..1] InitializesThread				
!<1-1>				
[0..1] initializesThreadWithRunnable	Yes	9	33.33%	Yes
<1-1>				
[0..1] this				
[1..1] implementsRunnable				
[0..1] helper				
[0..1] variable (String)				
[0..1] runnableField				
[1..1] typedRunnable				
[1..1] name (String)				
[1..1] initialized	No	N/A	N/A	N/A
[0..1] initializesWithThreadSubclass	Yes	2	40.00%	No
[1..1] name (String)				
[1..1] overridesRun				
[1..1] extendsThread				
[1..1] nullifiesThread	Yes	5	26.32%	Yes
[0..*] singleTaskThread	No	N/A	N/A	N/A
<1-1>				
[0..1] runnable				
[0..1] runnableField				
[1..1] typedRunnable				
[1..1] name (String)				
[1..1] initialized	No	N/A	N/A	N/A
[0..*] parameter	Yes	41	25.00%	No
[0..*] name				

Table 5.1: Result of phase 2 of the study for the Java Applet framework

5.3.2 Results and Analysis of Phase 2

Table 5.1 presents the result for the Java applets and Table 5.2 presents the result for the Struts Application.

As in the result from the first study, the first column in both figures shows the features in each FSML with the row of the features that can have variable location in

Partial Applet FSML	>0 Instances	# of Instances without Defaults	% of Instances without Defaults	Can Code Entangle with Application Logic?
[0..*] ActionImpl				
![1..1] name (String)				
[0..1] package (String)				
[1..1] qualifiedName (String)				
[0..1] local				
![1..1] extendsAction				
[0..1] extendsDispatchAction				
[0..*] actionMethod (String)				
[0..1] overridesExecute				
[0..*] forwards	Yes	227	74.43%	Yes
[1..1] name (String)				
[1..1] forward				
<1-2>				
[0..1] localForward (ForwardDecl)				
[0..1] globalForward (ForwardDecl)				
[0..*] inputForwards	Yes	2	20.00%	Yes
[1..1] name (String)				

Table 5.2: Result of phase 2 of the study for the Apache Struts framework

its code pattern highlighted. The next three columns in both figures are calculations directly derived from the result of the first phase of the study. The first of these three columns shows whether the feature has more than one feature instance in the sample set of applications. The other two of the three columns show the number and the percentage of feature instances that are not handled by the default location, which we defined in phase 1 of the study to be the method the code patterns of the feature instances appear in more than 20% of the time. Framework-based code assistant has a clear advantage over forward engineering from the model editor in these feature instances, since framework-based code assistant allows code to be generated in any location as the user programs, but for forward engineering from the model editor, the code patterns for these instances would require manual code movement in the code editor. The last column shows, from manual code inspection, whether code patterns for the feature instances entangle with other application logic code. The implication for the features that do is that even feature instances with a default location will require further code customization, namely, moving the code to a specific line within a method, since forward engineering from the model editor will not be able to place the generated code in the expected location without fully understanding the user's application logic code. The rest of this section describes the findings during manual code inspection of the feature instances to see whether the code patterns for the feature instances of each feature entangle with application logic code.

For the Applet FSML, the feature `showStatus` always entangles with the application logic since, as described previously, depending on the application logic, different messages are displayed in the status window. Figure 5.3(a) gives an example of such a case. Depending on the value of r, different messages are displayed to the user in the status window.

The features for the register listener and deregister listener calls are always just a single method call that has no dependency on the application logic code. There are a few examples, however, when the register listener calls in an applet are to register listeners not for the applet itself, but for a component used in the applet.

The feature `initializesThreadWithRunnable` corresponds to an assign-new call. A common micropattern with this feature is that the variable that the thread is assigned to is often checked to make sure that it is null and/or the thread is not alive before assigning it to the new thread variable again. Also, sometimes, a thread is only initialized under other special circumstances that are application specific. One such example is shown in Figure 5.3(b). The thread is only initialized if the command received is “blink”.

The feature `initializesThreadWithSubclass` is generally not entangled with the application logic and is just assigned to a variable that is subsequently used. Therefore, forward engineering from the model editor can safely generate the call at the beginning of the method, so that application logic code can use the variable later on in the method.

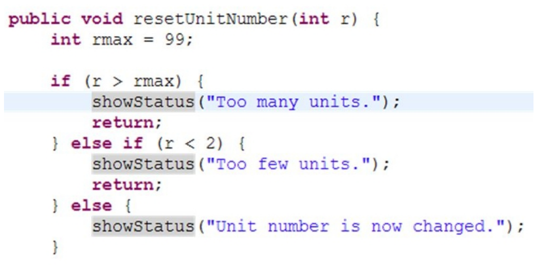
The feature `nullifiesThread` is usually a single assign-to-null statement. Oftentimes, like `initializesThreadWithRunnable`, the thread is usually checked to see that it is not already null before assigning to null. Also, one example of `nullifiesThread` involves nullifying the thread when an application-specific exception arises (Figure 5.3(c)).

The feature `parameter` is usually entangled with the application logic, but it is always possible to refactor the code such that the getParameter call is first assigned to a variable in the beginning of a method body before using it with the application logic code in the rest of the method body. Hence, we consider it as not entangled, and no further customization is needed when forward engineering the concept from the model editor. Figure 5.3(e) shows an example from the set of sample applications where the getParameter calls can be refactored into variable assignment statements in the beginning of the method that the code for the framework concept resides in.

For the Struts FSML, both the `forwards` and `inputForwards` feature can entangle with the application logic code when different pages are forwarded depending on the application logic. Figure 5.3(d) shows such an example from the set of sample applications.

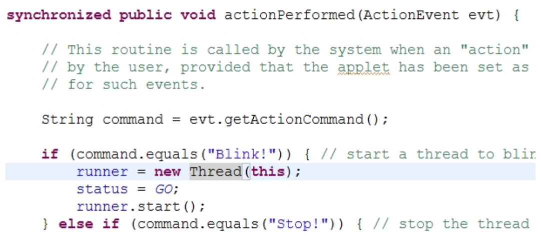
5.4 Threats to Validity

There are a few threats to validity in the evaluation presented in this chapter. Firstly, the feature instances that are used in the analysis are from framework-specific models that are reverse engineered from application code. Therefore, the accuracy of the code queries that are used in reverse engineering greatly affects the feature instances in the evaluation. However, a previous study [Antkiewicz et al., 2008] has shown the code queries to be of very high precision and recall, so that the feature instances used in the evaluation should be a relatively accurate representation of the actual feature instances in the sample application code. Secondly, we did not consider coding style in the evaluation, which would likely affect the evaluation result. More specifically, the location of a code pattern can also be affected by



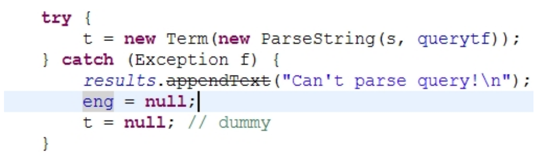
```java
public void resetUnitNumber(int r) {
    int rmax = 99;

    if (r > rmax) {
        showStatus("Too many units.");
        return;
    } else if (r < 2) {
        showStatus("Too few units.");
        return;
    } else {
        showStatus("Unit number is now changed.");
    }
```

(a) showStatus



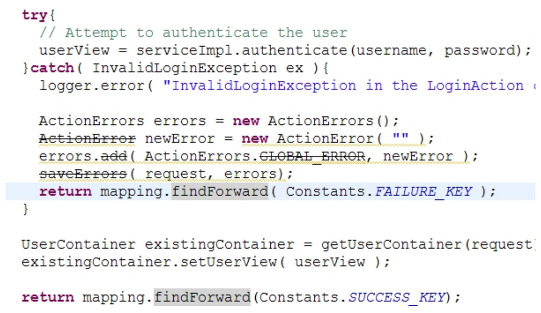
```java
synchronized public void actionPerformed(ActionEvent evt) {

    // This routine is called by the system when an "action"
    // by the user, provided that the applet has been set as
    // for such events.

    String command = evt.getActionCommand();

    if (command.equals("Blink!")) { // start a thread to blir
        runner = new Thread(this);
        status = GO;
        runner.start();
    } else if (command.equals("Stop!")) { // stop the thread
```

(b) initializesThreadWithRunnable



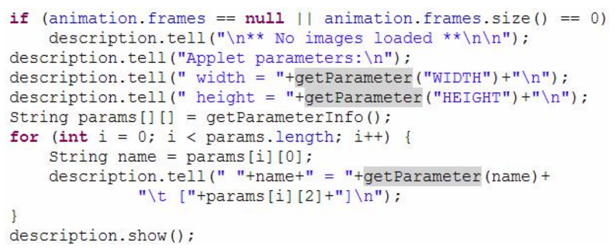
```java
try {
    t = new Term(new ParseString(s, querytf));
} catch (Exception f) {
    results.~~appendText~~("Can't parse query!\n");
    eng = null;
    t = null; // dummy
}
```

(c) nulliflesThread



```java
try{
  // Attempt to authenticate the user
  userView = serviceImpl.authenticate(username, password);
}catch( InvalidLoginException ex ){
  logger.error( "InvalidLoginException in the LoginAction

  ActionErrors errors = new ActionErrors();
  ~~ActionError~~ newError = new ActionError( "" );
  errors.~~add~~( ActionErrors.~~GLOBAL_ERROR~~, newError );
  ~~saveErrors~~( request, errors);
  return mapping.findForward( Constants.FAILURE_KEY );
}

UserContainer existingContainer = getUserContainer(request
existingContainer.setUserView( userView );

return mapping.findForward(Constants.SUCCESS_KEY);
```

(d) forwards



```java
if (animation.frames == null || animation.frames.size() == 0)
    description.tell("\n** No images loaded **\n\n");
description.tell("Applet parameters:\n");
description.tell(" width = "+getParameter("WIDTH")+"\n");
description.tell(" height = "+getParameter("HEIGHT")+"\n");
String params[][] = getParameterInfo();
for (int i = 0; i < params.length; i++) {
    String name = params[i][0];
    description.tell(" "+name+" = "+getParameter(name)+
            "\t ["+params[i][2]+"]\n");
}
description.show();
```

(e) parameter

Figure 5.3: Framework feature code entangled with application logic code

coding style and so increases the advantage the framework-based code assistant has over forward engineering from the model editor, since the framework-based code assistant allows code to be generated at any valid location.

Several factors also affect the generalization of the result: the sample applications, the concepts the FSMLs cover, and the frameworks that we used. We have put in efforts to reduce the threats of validity from the sample applications by selecting a mix of applications shipped with the framework and open source applications. Biases in the concepts the FSMLs cover are also minimized by the fact that the FSMLs are created prior to the inception of the framework-based code assistant in this thesis. Although the two frameworks that are chosen are widely popular frameworks, an investigation on the generalization of the result to other frameworks remains future work.

Chapter 6

Conclusion

Object oriented frameworks are useful and popular in application development but are known to be difficult to use. With the formalization of framework concepts and constraints in a framework-specific modeling language, advanced context-sensitive code assistance, guided by a framework-specific model expressed using the language, can be provided to the user directly in the code editor. The code assistance aids the user in implementing framework concepts with the consideration of framework constraints. Compared to code generation from the model editor, our framework-based code assistant: 1) avoids the need to synchronize changes between the model and the code, 2) automatically forwards all mandatory subfeatures recursively, 3) allows the user to work directly in the code that avoids the switching between the model editor and the code editor, since working in the code editor is inevitable given that FSML is not a general purpose modeling language, and 4) reduces the need to perform code customization that occurs when generating code at fixed location from the model editor. The evaluation in this thesis characterizes features into anchor, non-floating, semi-floating, and free-floating features and finds that a number of features are semi-floating or free floating and entangles with the application logic code. When performing code generation from the model editor, these features require moving the generated code to the desired location, but our framework-based code assistant does not require such code customization.

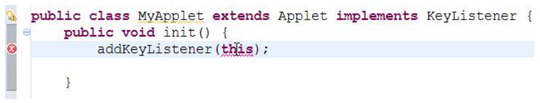
6.1 Future Work

Several possible directions for future work follow:

User study and further evaluation. While we have evaluated our framework-based code assistant by comparing it against forward engineering from the model editor where code is generated at fixed locations, a user study to evaluate the usability of our approach remains future work. For example, we could design a user study to compare the framework-based code assistant described in this thesis against programming-language based code assistants found in traditional IDEs in order to assess productivity improvements. More FSMLs can also be designed and tested with the framework-based code assistant to evaluate the usefulness of the assistant across a larger variety of frameworks.

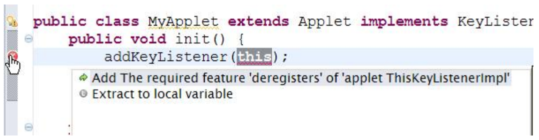
Framework-based code assistants as recommender systems. Framework-specific modeling languages can be extended to record the likelihood of a feature by expressing the languages as probabilistic feature models [Czarnecki et al., 2008]. With support for probabilities in the FSMLs, recommendations can be suggested to the user in the form of quick assists or any new user interface that suggests the recommendations with associated confidence levels as the user programs. Also, much like how Firefox's keyword support in its "Awesomebar" ranks the result by frequency and recency of visits to the page, and how some of the related work in the literature described in section 2.2 ranks proposals according to the likelihood of a proposal, such as Mylyn [Kersten and Murphy, 2005], FrUiT [Bruch et al., 2006], and Improving Autocompletion using Program History [Robbes and Lanza, 2008], probabilistic feature models can also be used to help rank proposals in our framework-based code assistant. Proposals that are more likely to be used or relevant can be ranked higher in the proposal list in the code assistant.

Other framework-specific IDE extensions. Framework-based code assistants can be seen as a type of framework-specific IDE extensions. We have explored this topic and the use of framework-specific models in supporting different framework-specific IDE extensions in [Lee et al., 2008]. For example, figure 6.1(a)–6.1(c) show an example of framework-based error identification with quick fixes by performing model validation, and figure 6.1(d) shows an example of a framework-based content outline. Future work can also include framework support in the IDE using framework-specific models for other phases of software development, including compilation, debugging, testing, deployment, and version control, such that a full framework-based IDE can be realized.



```
public class MyApplet extends Applet implements KeyListener {
    public void init() {
        addKeyListener(this);

    }
```

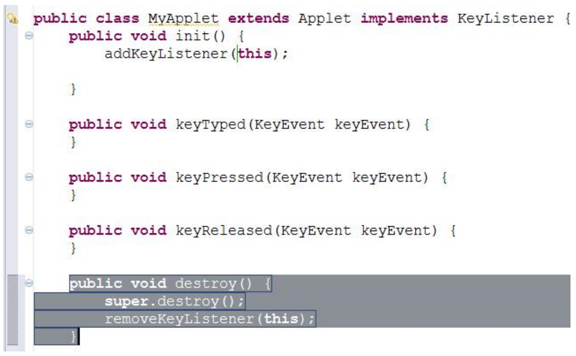
(a) Framework-Based Error Identification



```
public class MyApplet extends Applet implements KeyLister
    public void init() {
        addKeyListener(this);
```

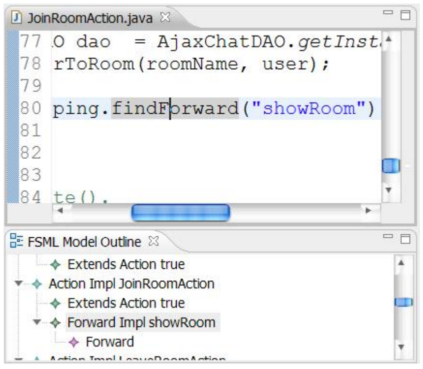
Add The required feature 'deregisters' of 'applet ThisKeyListenerImpl'
Extract to local variable

(b) Framework-Based Quick Fix



```
public class MyApplet extends Applet implements KeyListener {
    public void init() {
        addKeyListener(this);

    }

    public void keyTyped(KeyEvent keyEvent) {
    }

    public void keyPressed(KeyEvent keyEvent) {
    }

    public void keyReleased(KeyEvent keyEvent) {
    }

    public void destroy() {
        super.destroy();
        removeKeyListener(this);
    }
```

(c) Resulting Code after Applying Framework-Based Quick Fix



```
O dao  = AjaxChatDAO.getInst
rToRoom(roomName, user);

ping.findForward("showRoom")




te().
```

(d) Framework-Based Content Outline

Figure 6.1: Other framework-based IDE extensions

APPENDICES

Appendix A

Mapping Types & Constraints in Framework-Specific Modeling Languages

The following tables are from [Antkiewicz, 2008] and are provided below for convenience only.

Table A.1: Mapping types for structural code patterns for XML

Structural Pattern Expression	**Structural Element(s) Matched**
p xmlDocument: h	matches a XML document at path h in project p
d xmlElement: n	matches a root XML element with the name n of the XML document d
e xmlElements: p	matches a XML elements at path p relative to XML element e
e xmlAttribute[: n]	matches a value of XML attribute called e of the XML element e. The parameter n is optional: the name of the feature is used in its absence
e xmlElementValue	matches a value of the XML element e

Table A.2: Constraints

Structural Pattern Expression	**Structural Element(s) Matched**
where: f contains: g	true if the value of the feature f contains the value of the feature g
where: f equalsTo: g	true if the value of the feature f is the same as the value of the feature g
valueEqualsTo: f value: v	true if the value of the feature f is the same as the value v
andParentIs: f	true if the the parent of the reference value is feature f
valueOf: f class: p	matches the value of the feature f which is the parent of feature p
constraint: f implies: g	false if the feature f is present and the feature g is missing. True otherwise (implication)

Table A.3: Constraints and parameters for forward engineering

Constraint	**Meaning**
subsumedBy: f	specifies that the code transformation for a feature should not be executed if the feature f is present
Parameter	**Meaning**
initializer: i	parameter i specifies whether a field assignment should be created in the field's initializer
location: s	specifies that the code should be inserted in the body of the method with the signature s
position: p	parameter p specifies whether the code should be inserted at the beginning (p=before) or at the end (p=after) of the method's body
receiverExpr: e	specifies the expression e that should be inserted as a receiver of a method call

Table A.4: Mapping types for structural code patterns for Java

Structural Pattern Expression	**Structural Element(s) Matched**	**Abbreviation**
project	matches a Java project	project
p projectName	matches the name of the project p	projectName
class	matches a Java class	class
c fullyQualifiedName	matches the fully qualified name of the class c	fqName
c className	matches the simple name of the class c	className
c qualifier	matches the qualifier (package name) of the class c	qualifier
c assignableTo: t [concrete: r] [local: p]	matches if objects of the class c are assignable to the type t. The optional parameter r specifies that only concrete classes should be matched. The optional parameter p specifies that only source types of the project p should be matched	assignable
c isLocal: p	matches if the class c is a source class of the project p	isLocal
field	matches a field	field
f fieldName	matches the name of the field f	fieldName
f fieldOfType: t	matches if objects of the type t are assignable to the field f	fieldOfType
c methods: s	matches methods with signature s that are implemented or overridden by the class c. The signature may contain * for the method name to match any method name	methods
c allMethods: s	matches methods with signature s that are implemented, overridden or inherited by the class c. The signature may contain * for the method name to match any method name	allMethods
x annotatedWith: t	matches a Java 5 annotation of type t placed on element x. The element x can be a class, a method, or a field	annotatedWith
a attribute: n	matches the value of attribute called n of a Java 5 annotation a	attribute
a hasNoAttribute	matches if the annotation a has no attributes (i.e., the annotation is a *marker* annotation)	noAttribute
mc argumentIsThis: i class: c	matches if the i^{th} argument of the method call mc is a `this` literal assignable to the class c.	argIsThis
mc argumentIsField: i [sameAs: f]	matches if the i^{th} argument of the method call mc is a field. Parameter f is optional and specifies a constraint that the matched field must be the same as the field that the feature `f` corresponds to.	argIsField
mc argumentIsNew: i signature: s	matches if the i^{th} argument of the method call mc is a constructor call of the signature s	argIsNew
mc argumentIsVariable: i	matches if the i^{th} argument of the method call mc is a variable	argIsVar

Table A.5: Mapping types for behavioural code patterns for Java

Behavioural Pattern Expression	**Run-time Event Pattern(s) Matched**	**Abbrev.**
c callsTo: s receiver: r [statement: s]	matches method calls to methods with the signature s received by objects assignable to the type r in the control flow of instances of the class c. The optional parameter s specifies whether only method calls which are individual statements should be matched.	callsTo
	`callsTo($c o): call($s) && target($r) && cflow(execs(o))`	
c callsReceived: s	matches method calls to methods with the signature s received by objects assignable to the class c	callsRec
	`callsRec($c o): call($s) && target(o)`	
mc valueOfArg: i	matches run-time values of the i^{th} argument of the method call mc	argVal
	`argVal(): $mc && args(.., $i, ..)`	
c argument: i ofCall: mc_1 sameAsArg: j ofCall: mc_2	matches if the i^{th} argument of the method call mc_1 points to the same object as the j^{th} argument of the method call mc_2, in the control flow of objects of the class c	argSameObj
	`argSameObj($c o): $argVal(mc2, j) && dflow[j, i] ($argVal(mc1, i)) && execs(o)`	
c methodCall: mc_1 before: mc_2	matches if in the control flow of instances of the class c, the method call mc_1 occurs at least once before the occurrence of method call mc_2	before
	`before($c o): execs(o) && ($mc1+ $mc2)`	
m returnedObjectTypes: c	matches all possible types of the objects returned by the method m from the point of view of the class c that implements, overrides, or inherits m	retTypes
	`retTypes(): execution($m) && returnTypes() && this($c)`	
f assignedNull	matches assignments to the field f with the `null` value	assignNull
	`assignNull(Object o): set($f) && args(o) && if(o == null)`	
f assignedNew: cs [subtypeOf: t]	matches assignments to the field f with an object returned by a constructor call with the signature cs. The optional parameter subtypeOf specifies that only constructor calls that create instance of the subtype of the type t should be matched	assignNew
	`assignNew(Object o): set($f) && args(o) && dflow[o, i] (call($cs) && returns(i))`	
Helper Definitions	matches executions of methods in instances of class c	
	`execs($c o) : execution(* *(..)) && this(o);`	

Appendix B

Code Queries for Reverse Engineering

The following tables are from [Antkiewicz, 2008] and are provided below for convenience only.

Code Query Abbrev.	**Query Expression** / **Result**
getCallsWH*	c getCallsInHierarchy: s receiver: r
	a set of method calls with the signature s within the bodies of the class c and its superclasses, such that the receiver of each call is assignable to the type r
getCallsCF	c getCallsCFlow: s receiver: r
	a set of method calls with the signature s in the control flow of every implemented, inherited, and overridden method of the class c, such that the receiver of each call is assignable to the type r

Table B.1: Code queries for the *callsTo* mapping type

Code Query Abbrev.	**Query Expression** / **Result**
getCallsRec*	c getCallsReceived: s
	a set of method calls with the signature s, such that the receiver of each call is assignable to the type c
getCallsRecTI	c getCallsReceivedTI: s
	a set of method calls with the signature s, such that the receiver of each call is assignable to the type c. In the case when the type of the receiver is more general then the type c, the query traverses the receiver's dataflow graph backwards to infer its more specific type

Table B.2: Code queries for the *callsRec* mapping type

Code Query Abbrev.	**Query Expression** / **Result**
getArgValLC*	mc getArgValLiteralConstant: i
	value of the i^{th} argument of the method call mc retrieved from a `static final` variable or a literal
getArgValCP	mc getArgValConstantProp: i
	set of values of the i^{th} argument of the method call mc retrieved using interprocedural constant propagation limited in scope to the class that contains the called method
getArgValPE	mc getArgValPartialEval: i
	set of values of the i^{th} argument of the method call mc retrieved using partial evaluation

Table B.3: Code queries for the *argVal* mapping type

Code Query Abbrev.	**Query Expression** / **Result**
argIsThis*	c thisAsArgument: i ofCall: mc_1 andArg: j ofCall: mc_2
	true iff both the i^{th} argument of the method call mc_1 and the j^{th} argument of the method call mc_2 are the literal `this` and the resolved type of the literal is class c
argIsPrvFieldAO	c prvFieldAsArgument: i ofCall: mc_1 andArg: j ofCall: mc_2 givenCSeq: cs
	true iff both the i^{th} argument of the method call mc_1 and the j^{th} argument of the method call mc_2 are references to the same private field of class c whose value has been assigned once before both calls

Table B.4: Code queries for the *argSameObj* mapping type

Code Query Abbrev.	**Query Expression** / **Result**
isBeforeWH*	c is: mc_1 before: mc_2 inHierarchyGivenCSeq: cs
	true iff the method calls mc_1 and mc_2 are located within the bodies of callback methods m_1 and m_2, respectively, such that the method m_1 occurs before the method m_2 in the callback sequence cs OR true iff mc_1 occurs before mc_2 in the cflow of the method m_1 if $m_1 = m_2$. Methods m_1 and m_2 can be any implemented, inherited or overridden methods of the class c
isBeforeCF	c is: mc_1 before: mc_2 inCFlowGivenCSeq: cs
	true iff the method calls mc_1 and mc_2 occur in the control flows of callback methods m_1 and m_2, respectively, such that the method m_1 occurs before the method m_2 in the callback sequence cs OR true iff mc_1 occurs before mc_2 in the cflow of the method m_1 if $m_1 = m_2$. Methods m_1 and m_2 can be any implemented, inherited or overridden methods of the class c

Table B.5: Code queries for the *before* mapping type

Code Query Abbrev.	**Query Expression** / **Result**
getRetTypesWS*	m returnStmsWithinAndSuper: c
	a set of types of objects returned by the method m (excluding Object) retrieved from type bindings of return statements within the body of the method, including bodies of super methods if called. The type of the returned literal `this` is interpreted as class c
getRetTypesMST	m returnStmsMostSpecificType: c
	a set of types of objects returned by the method m (excluding Object) retrieved from return statements, inferring the most specific type in the data flow of each returned object. The type of the returned literal `this` is interpreted as class c

Table B.6: Code queries for the *retTypes* mapping type

Code Query Abbrev.	**Query Expression** / **Result**
getAssgnNew*	f getAssignedNew: cc
	a set of assignments to the field f with the constructor call cc

Table B.7: Code queries for the *assgnNew* mapping type

Code Query Abbrev.	**Query Expression** / **Result**
getAssgnNull*	f getAssignedNull
	a set of assignments to the field f with the `null` literal

Table B.8: Code queries for the *assignNull* mapping type

Appendix C

Code Transformations for Forward Engineering

The following tables are from [Antkiewicz, 2008] and are provided below for convenience only.

Code Transformations for Structural Patterns

Tables C.1-C.8 present code transformations for structural patterns from Table A.4. Code transformation `addMethod` is also used for the mapping type `allMethods`.

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters**
	Result
addClass	p addClass: n [in: q] | q=”
	creates a compilation unit with a class declaration named n in package q. Retrieves values of the parameters n and q from subfeatures with mapping types `className` and `qualifier` or `fullyQualifiedName`

Table C.1: Code transformation for the *class* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters**
	Result
addAssignableTo	c addAssignableTo: t [concrete: e] | e=true
	If t is an interface, adds a `c implements t` superinterface declaration or adds t to the existing list of implemented interfaces. If t is a class, adds a `c extends t` superclass declaration. If e=true, adds implementations of the unimplemented methods of the superinterface or an abstract superclass

Table C.2: Code transformation for the *assignableTo* mapping type

Code Transformations for Behavioural Patterns

Tables C.9-C.15 present code transformations for behavioural patterns from Table A.5. There is no code transformation for the mapping type `before`. It is the

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters** / **Result**
addField	*c* addField: *n* ofType: *t*
	adds a field declaration in the class *c* named *n* of type *t*. Retrieves values of the parameters *n* and *t* from subfeatures with mapping types `fieldName` and `fieldOfType`, respectively

Table C.3: Code transformation for the *field* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters** / **Result**
addMethod	*c* addMethod: *s* [name: *n*]
	adds a method declaration of signature *s* in the class *c*. If method name *n* is given, replaces the name from the signature *s* with *n*. If the signature contains * for the method name, the parameter *n* is mandatory. Retrieves the value of the parameter *n* from a feature with the mapping type `methods`

Table C.4: Code transformation for the *methods* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters** / **Result**
addArgIsThis	*mc* addArgIsThis: *i*
	adds the literal `this` as the i^{th} argument of the method call *mc*

Table C.5: Code transformation for the *argIsThis* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters** / **Result**
addArgIsField	*mc* addArgIsField: *i* [sameAs: *f*]
	adds the field *f* as the i^{th} argument of the method call *mc*. If *f* is not specified, retrieves the name of the field from a subfeature with the mapping type `fieldName`

Table C.6: Code transformation for the *argIsField* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters** / **Result**
addArgIsNew	*mc* addArgIsNew: *i* signature: *s*
	adds the constructor call of the signature *s* as the i^{th} argument of the method call *mc*. Creates an anonymous subclass if necessary and adds implementation of unimplemented methods

Table C.7: Code transformation for the *argIsNew* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters** / **Result**
addArgIsVar	*mc* addArgIsVar: *i* name:*n* signature: *s*
	adds a variable called *n* as the i^{th} argument of the method call *mc*. Adds the variable declaration and initializes the variable with a constructor call of the signature *s*. Retrieves the value of the parameter *n* from the feature with the mapping type `argIsVar`

Table C.8: Code transformation for the *argIsVar* mapping type

responsibility of the FSML designer to specify target methods such that the first method call occurs before the second one.

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters**
	Result
addCallTo	c addCallTo: s [receiverExpr: r] location: l [position: p] | r=”, p=after
	creates a method call to a method with the signature s with the receiver expression r in the method of signature l of the class c at the position $p \in$ {before, after}

Table C.9: Code transformation for the *callsTo* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters**
	Result
addCallRec	c addCallTo: s [receiverExpr: r] location: l [position: p] | r=”, p=after
	creates a method call to a method with the signature s with the receiver expression r in the method of signature l of the class c at the position $p \in$ {before, after}

Table C.10: Code transformation for the *callsRec* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters**
	Result
addArgVal	mc addArgVal: i [values: v]
	adds values of the i^{th} argument of the method call mc. Adds a literal for a single value. For multiple values create a variable and multiple assignments with values v

Table C.11: Code transformation for the *argVal* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters**
	Result
addArgSameObj	c addArgument: i ofCall: mc_1 andArg: j ofCall: mc_2
	adds the literal `this` that resolves to class c as both the i^{th} argument of the method call mc_1 and the j^{th} argument of the method call mc_2

Table C.12: Code transformation for the *argSameObj* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters**
	Result
addRetTypes	m addReturnStms: c ifkey: i
	for each type t from the list of types c adds a return statement to the method m returning an object of that type. Adds each return statement at the beginning of the method and preceeds each return statement with an if statement of the form `if (t.class.equals(x)))`, where t is the type and x is the name of the method's ith parameter.

Table C.13: Code transformation for the *retTypes* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters**
	Result
addAssgnNew	f addAssignedNew[: cc] [initializer: i] [location: l position: p] [subtypeOf: t] | i=false, p=after
	Two variants: (1) adds an assignment to the field f with the constructor call of the signature cc. (2) adds an assignment to the field f with the constructor call to the default constructor of a subclass of the type t. Retrieves values of the parameters n and q from subfeatures with mapping types `className` and `qualifier` or `fullyQualifiedName`. Regardless of the variant: if i=true, adds the assignment in field's initializer; otherwise, adds the assignment in the method l at position $p \in$ {before, after}. Creates an anonymous subclass if necessary and adds implementation of unimplemented methods

Table C.14: Code transformation for the *assgnNew* mapping type

Code Transformation Abbrev.	**Transformation Expression | Default values for optional parameters** / **Result**
addAssgnNull	f getAssignedNull location: l [position: p] | p=after
	adds an assignment to the field f with the `null` literal in the method l at position p $\in$ {before, after}

Table C.15: Code transformation for the *assgnNull* mapping type

Appendix D

Applications used in Evaluation

The following is a list of applications used to do the experiment and is the same list as the one used to evaluate the precision and recall of code queries in reverse engineering from [Antkiewicz, 2008]. The list is provided below for convenience only.

D.1 Struts

The set applications used in the study consists of 6 applications.

- 2 example Struts applications shipped with the framework: Cookbook and Mailreader 1.3.8;
- 1 large, open-source, production quality application: Apache Roller Weblogger 3.1; and
- 3 small, open-source applications: Ajax Chat 1.2, Beer4all, and Pools 2.5.

D.2 Applets

The set of applets used in the study consists of 84 applets. The applets are divided in a few groups.

- 20 applets examples shipped with Sun's JDK;
- 51 applets obtained from the internet by George Fairbanks and used in his study of design fragments: ANButton, Antacross, AquaApplet, BlinkingHelloWorld2, BrokeredChat, Bsom1,ButtonTest, Client, ConsultOMatic, ContextTestExecutor, Demographics, DotProduct, Envelope, ErrorMessage, Fireworks, FormelnApplet, GammaButton, Geometry, HelloTcl, HitMeter,

HmFetcher, Iagttager, InspectClient3, JScriptExample, KeyboardAndFocus-Demo, LinProg, MarchingAnts, MouseDemo, MyApplet, MyApplet, Nick-Cam, ScatterPlotApplet, Scope, SilentThreat, SimplePong, SimpleSunApplet, SmtpApp, SuperApplet, SwatchITime, hyperbolic.Test, TetrisApp, URL-ExampleApplet, ungrateful.Ungrateful, ungrateful.OutPanel, UrcrcCalendar, VeChat, notprolog.WPrologGUI, notprolog.WProlog, WebStart, YmpyraAppletti, CaMK;

- 8 applets by R. Bowles: Bioquiz, Calculator, Crystal, Frogs, LightRays, Mandel, Mastermind, Starscape; and
- 5 applets from three open-source project (SourceForge): JugglingLab (3 applets), snirc 1.0 (1 applet: Chat), sudoku (1 applet: Main).

References

Michal Antkiewicz. *Framework-Specific Modeling Languages*. PhD thesis, University of Waterloo, 2008. (Cited on pages 1, 2, 12, 13, 15, 16, 43, 46, 49, and 53.)

Michal Antkiewicz and Krzysztof Czarnecki. Design space of heterogeneous synchronization. In *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 3–46, 2007. (Cited on page 9.)

Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *MoDELS 2006: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*, pages 692–706, 2006. (Cited on pages 1 and 12.)

Michal Antkiewicz, Thiago T. Bartolomei, and Krzysztof Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *ASE '07: Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 214–223, 2007.

Michal Antkiewicz, Thiago Tonelli Bartolomei, and Krzysztof Czarnecki. Fast extraction of high-quality framework-specific models from application code. *Journal of Automated Software Engineering*, 2008. (Cited on pages 1, 15, 30, and 37.)

Michal Antkiewicz, Krzysztof Czarnecki, and Matthew Stephan. Engineering of framework-specific modeling languages. *Transactions of Software Engineering,Special Issue on Language Engineering*, 2009. (Cited on page 13.)

ATLAS. *ATLAS MegaModel Management (AM3) Home page*, 2008. `http://www.eclipse.org/gmt/am3/`. (Cited on page 11.)

Elizabeth Bjarnason and Görel Hedin. Tool support for framework-specific language extensions. In *Object-Oriented Technology, European Conference on Object-Oriented Programming (ECOOP) Workshop*, pages 129–132, 1997. (Cited on page 11.)

J. Bosch, P. Molin, M. Mattsson, and P.O. Bengtsson. Framework - problems and experiences. In M. Fayad, D. Schmidt, and R. Johnson, editors, *Building Application Frameworks*. John Wiley, 1999.

Marcel Bruch, Thorsten Schäfer, and Mira Mezini. FrUiT: Ide support for framework understanding. In *Eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 55–59, 2006. (Cited on pages 10 and 40.)

David Carlson. *Eclipse Distilled*. Eclipse. Addison-Wesley Professional, 2005. (Cited on page 5.)

Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton Jr. IMP: A metatooling platform for creating language-specific IDEs in Eclipse. In *ASE '07: Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 485–488, 2007. (Cited on page 11.)

Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. (Cited on page 12.)

Steve Cook. The domain-specific IDE, 2008. Keynote at Code Generation.

K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models: There and back again. *SPLC '08. 12th International*, pages 22–31, Sept. 2008. (Cited on page 40.)

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. Addison-Wesley Publishing Co., 2000. (Cited on page 12.)

Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003. (Cited on page 8.)

Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration using feature models. In *SPLC '04: Proceedings of the 3rd international conference on Software Product Lines*, pages 266–283, 2004. (Cited on page 12.)

Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

Java Emitter Templates Component. Eclipse Foundation, 2007. `http://www.eclipse.org/modeling/m2t/?project=jet`. (Cited on page 8.)

Help -Eclipse SDK. Eclipse Foundation, 2007a. `http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.jdt.doc.user/concepts/cquickassists.htm`. (Cited on page 8.)

Help -Eclipse SDK. Eclipse Foundation, 2007b. `http://help.eclipse.org/help33/topic/org.eclipse.jdt.doc.user/concepts/cquickfix.htm`. (Cited on page 7.)

Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The TOPCASED project: a toolkit in open source for critical aeronautic systems design. In *ERTS '06: Embedded Real Time Software*, 2006. (Cited on page 11.)

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. (Cited on page 9.)

Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, and Jukka Viljamaa. Generating application development environments for Java frameworks. In *GCSE '01: Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering*, pages 163–176, 2001. (Cited on page 11.)

Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wąsowski. Guided development with multiple domain-specific languages. In *MoDELS 2007: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, 2007.

Abbas Heydarnoori, Krzysztof Czarnecki, and Thiago Tonelli Bartolomei. Supporting framework use via automatically extracted concept-implementation templates. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, July 2009.

Rosco Hill and Joe Rideout. Automatic method completion. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 228–235, 2004. (Cited on page 10.)

Reid Holmes and Gail C Murphy. Using structural context to recommend source code examples. In *In ICSE Š05: Proceedings of the 27th international conference on Software engineering*, pages 117–125. ACM Press, 2005. (Cited on page 10.)

D. Hou and H.J. Hoover. Using scl to specify and check design intent in source code. volume 32, pages 404–423, June 2006.

Daqing Hou. *FCL: Automatically Detecting Structural Errors in Framework-Based Development.* PhD thesis, University of Alberta, 2004.

Daqing Hou, Kenny Wong, and H. James Hoover. What can programmer questions tell us about frameworks? In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 87–96, 2005. (Cited on page 1.)

Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. (CMU/SEI-90TR -21), 1990. (Cited on page 12.)

Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for IDEs. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, 2005. (Cited on pages 10 and 40.)

Douglas Kirk, Marc Roper, and Murray Wood. Identifying and addressing problems in object-oriented framework reuse. volume 12, pages 243–274, Hingham, MA, USA, 2007. Kluwer Academic Publishers. (Cited on page 1.)

Herman Hon Man Lee, Michal Antkiewicz, and Krzysztof Czarnecki. Towards a generic infrastructure for framework-specific integrated development environment extensions. In *2nd International Workshop on Domain-Specific Program Development (DSPD), in association with GPCE*, 2008. (Cited on page 40.)

Greg Little and Robert C. Miller. Translating keyword commands into executable code. In *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 135–144, New York, NY, USA, 2006. ACM. ISBN 1-59593-313-1.

Greg Little and Robert C. Miller. Keyword programming in Java. In *ASE '07: Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 84–93, 2007. (Cited on page 9.)

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61, 2005. (Cited on page 10.)

Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006. ISSN 0740-7459. (Cited on page 6.)

Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 742–745, New York, NY, USA, 2000. ACM. ISBN 1-58113-206-9. (Cited on page 9.)

openArchitectureWare. openArchitectureWare.org, 2008. `http://www.openarchitectureware.org`. (Cited on page 8.)

Thomas Reps and Tim Teitelbaum. The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48, 1984. (Cited on page 11.)

Romain Robbes and Michele Lanza. How program history can improve code completion. pages 317–326, Sept. 2008. doi: 10.1109/ASE.2008.42. (Cited on pages 10 and 40.)

Naiyana Sahavechaphan. Xsnippet: mining for sample code. In *OOPSLA '06: Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 413–430. ACM Press, 2006. (Cited on page 10.)

Shane Sendall and Jochen Küster. Taming model round-trip engineering. In *Proceedings of Workshop ŠBest Practices for Model-Driven Software Development*, 2004. (Cited on pages 2 and 9.)

Perdita Stevens. A landscape of bidirectional model transformations. In *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 408–424, 2007. (Cited on page 9.)

David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999. (Cited on page 12.)

XDoclet. XDoclet Team, 2005. http://xdoclet.sourceforge.net. (Cited on page 8.)

Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 164–173, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. (Cited on page 9.)