Hook and Template Coverage Criteria for Testing Framework-based Software Product Families

Raine Kauppinen, Juha Taina and Antti Tevanlinna

University of Helsinki
Department of Computer Science
P. O. Box 68 (Gustaf Hällströmin katu 2 B)
FIN-00014 UNIVERSITY OF HELSINKI
{Raine.Kauppinen, Juha.Taina, Antti.Tevanlinna}@cs.helsinki.fi

Abstract. In this work, we introduce the concepts of *hook* and *template coverage criteria* for integration testing of framework-based product families. These product family specific coverage criteria define how much functionality of the application code expanding an application framework has been covered with existing test suites. The criteria are simple, but relevant. We demonstrate the relation of the proposed and traditional criteria in an example.

1 Introduction

While there have been related large scale projects and case studies [1,2] and a framework for software product line practice has been developed [3], surprisingly little is written about testing of product families. The traditional object-oriented methods [4] for testing can be used, but there is also growing demand for a well-defined product family testing process and methodology including tool support [5]. One part of the work is to define coverages for estimating adequacy of testing in product family context.

We propose two new coverage criteria for framework-based product families: *hook coverage* and *template coverage*. They define how much functionality of the application code extending an application framework have been covered with existing test suites.

This work is organized as follows. Section 2 introduces framework-based software product families and their testing. Section 3 defines hook and template coverages. Section 4 shows an example of these coverages and finally, Section 5 concludes this article.

2 Framework-based Software Product Families and Testing

Object-oriented application frameworks provide a means to implement product families [6,7]. An application framework is a partial design and implementation for an application in a given domain. To create an application, missing functionality and, if required, new features must be added.

Frameworks are adapted by adding application specific functionality to the structure the framework provides [6]. Variation points open for customization in a framework are called *hot spots* while stable parts are called *frozen spots*. Hot spots are implemented as *hook classes* and frozen spots as *template classes*. A template class contains *template*

methods that use services of a hook class [7]. The hook class is abstract, so its *hook methods* must be implemented when the framework is extended.

Figure 1 shows an example of an application derivation from a framework. The left side of the figure shows the structure of the application. The framework defines interfaces with templates and hooks that are used to implement the application specific functionality of the product as shown in the center of the figure. When the template methods in the framework call abstract hook methods, virtual binding delivers the invocation to application specific code enabling variability. This is illustrated in Figure 1 using curved arrows between methods. The messages from a template method are targeted to a hook method never reaching it, but instead pointing to the real receiver – the hook method implementations as shown on the right side of the figure.

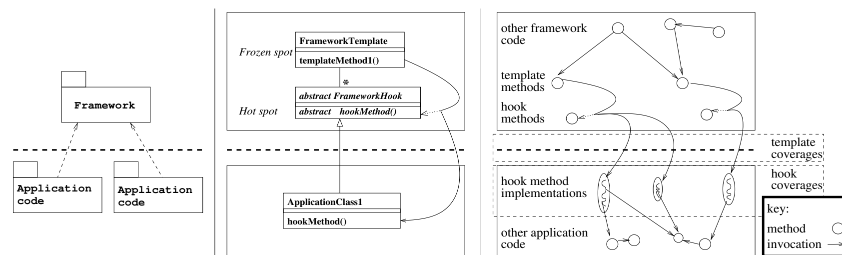

Fig. 1. An example of the structure of an application derived from a product family based on an object-oriented framework (on the left), the class level interface between the framework and the application specific code element extending it (in the center) and context of the hook and template coverages (on the right).

In a product family organization, different teams implement unit tests for the framework and the application code and, during testing of the framework, it is most convenient to use minimal stub implementations of the hot spots. Hot spots promote isolation in unit testing, because it is relatively easy to implement stubs and take them into use. Therefore, integration testing is crucial as the framework is first unit tested without complete knowledge of the future applications. Also, application-specific parts are unit tested assuming the framework is well-tested.

The framework interface requires new coverage criteria for integration testing to complement the traditional structural coverages. Traditional structural coverages have given a good indication of the integration testing coverage when there is no dynamic binding. For example, condition coverage subsumes coverage of the invocations in call graph of a program. But with dynamic binding, there may be several arcs in the graph for a method invocation in code and the indication of adequacy provided by traditional criteria is significantly weaker.

Many criteria have been defined to test polymorphism in object-oriented languages [4,8,9]. We aim to stronger integration testing criteria still retaining simplicity. The template coverage we propose is similar to existing criteria. The hook coverage is new and stronger than most existing criteria taking into account structural aspects in addition to the existence of a polymorphic binding. The results and application of the hook

coverages are based on widely used traditional coverages such as statement coverage. We also claim the hook and template coverages are simple to understand, which makes them easy to use and build tool support for.

3 Hook and Template Coverage Criteria

A hook method reference is a method call from a template class to a hook method implementation residing in adapting code. A test case *goes through* a hook method reference if the test case invokes the reference; that is, if the test case executes the statement containing the method call from the template class to the hook method. A hook method reference is *tested* if it is gone through by one or more test cases and *not tested* otherwise. Furthermore, a test case *reaches* a hook method implementation if it goes through a hook method reference and if it executes some or all of the code in the method.

We define the *hook method coverage* HMC for a hook method implementation hm and the *template method coverage* TMC for a template method tm as follows:

Definition 1 *HMC(hm) = the structural coverage (for example the statement coverage) of hm provided by test cases that reach the method.*

Definition 2 *TMC(tm) = the number of tested hook references in tm divided by the number of the hook method references in tm.*

Next, we define the *hook class coverage* HCC for a hook class hc and the *template class coverage* TCC for a template class tc as follows:

Definition 3 *HCC(hc) = the number of covered structures divided by the number of coverable structures in the in hc. The structures to be counted depend on the structural coverage applied. For example, if statement coverage is used, these structures are statements.*

Definition 4 *TCC(tc) = the number of tested hook method references in tc divided by the number of hook method references in tc.*

The hook and template coverages can also be used to measure the coverage of frameworks or other collections of classes in an application by counting the structures or hook method references from them instead of single methods or classes. In a similar way, it is possible to measure the hook and template coverage of an application.

The hook coverage is best applicable at method, class or application level since hooks are application-specific. However, the template coverage is a product family level testing measure, because the same templates exist in every product of the family.

4 An Example and Analysis

We demonstrate the hook and template coverages using framework-based Osuma-application, which generates code for design patterns. In these examples, we deduce the

coverage values for simplicity. In real-life testing, an automatic calculation and analysis is a necessity. We have such an environment implemented in the *RITA environment* designed for product family testing [5].

The framework of Osuma contains hot spots for design patterns and querying information from the user to configure the structure to be created. An example of a frozen spot is the displaying of the questions in queries. Next, we analyze the testing of a template method that checks an answer to a query.

Figure 2 shows the relevant code of the template method containing three hook method invocations (See the underlined references to methods of the hook class reference *this.question* in the code) and the possible hook method implementations invoked by each hook reference. In the analyzed configuration, there are 15 potential hook references.

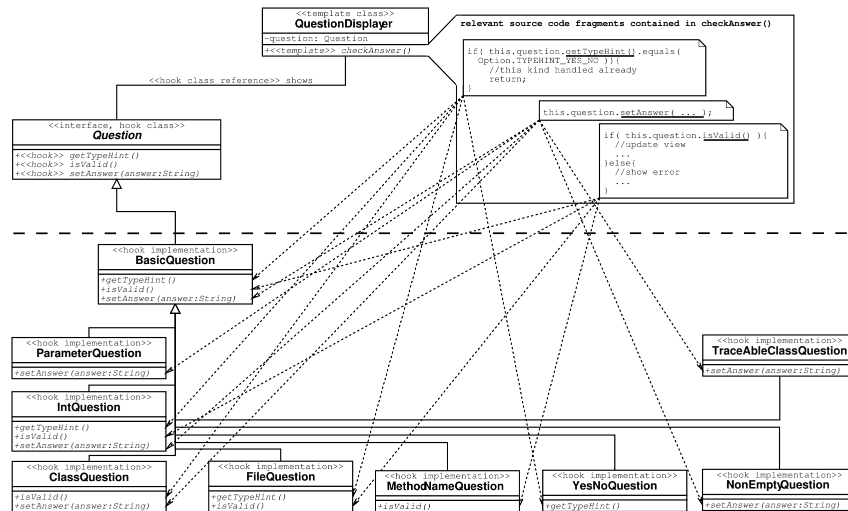

Fig. 2. The presentation of queries in the Osuma application. The framework-related functionality of a template method is illustrated by underlining the hook references in the code and connecting them to the relevant hook implementations.

Based on the information shown in the figure, it is possible to analyze the template coverage achieved when executing a test suite of QuestionDisplayer. The test suite contains three test cases. During first two test cases of the suite, the reference to the hook class contains an instance of IntQuestion. The invocations of the template method cover the three hook references to IntQuestion-hook implementation satisfying statement and branch coverage criteria in the last fragment of the code that appears in the upper right corner of the figure.

To satisfy the statement and branch coverage criteria in the first fragment of the code, a third test case is executed while the hook class reference contains an instance of YesNoQuestion. Thus, the references to IntQuestion and to YesNoQuestion are covered,

which yields the template coverage of 4/15. The coverage is quite low although statement and branch coverage criteria are satisfied. This is typical in presence of dynamic binding when testing objectives are defined by intra-method elements like branches.

Figure 3 contains the code of a implementation of the setAnswer-hook method in the Question-hook class. By analyzing execution of the test suite, we calculate the hook coverages of the method. The method is invoked two times. During both invocations the tested object is in a locked state and the first expression is never evaluated true. The *hook branch coverage* is 1/3 (including branching from the exception) and the *hook statement coverage* is 2/8. The binding from checkAnswer-method to setAnswer-method has been invoked, but the integration testing value of the interaction is limited. This may be noticed using the plain statement coverage. In the presence of intra-class re-entrance to hook methods and when using large test suites that mix unit and integration tests, it is unlikely that the inadequate integration would be evident using traditional coverage criteria. Notice that hook coverages always result lower than the coverages they are based on do.

```
public boolean setAnswer( String answer ){
  if( !this.locked ){
    try{
      Integer.parseInt( answer );
      this.valid = true;
      this.answer = answer;
      return true;
    }catch( NumberFormatException e ){
      //was not a number
      this.valid = false;
      this.message = "not a number";
    }
  }
  // no operation
  return false;
}
```

Fig. 3. Code of IntQuestion.setAnswer(String answer).

The implementor of the analyzed tests was one of the developers of the framework. In this case, the tester had no knowledge of the structures contained in the application specific classes. In the framework, the tests fulfilled the branch coverage criterion, but the poor hook branch coverage (1/3) indicates inadequate integration testing. Significant integration objectives were missed even in the elements that were integration tested. The guidance provided by the hook and template coverages is valuable in improving the structural testing in the case discussed.

5 Conclusions

We need product family specific testing methodology. One part of the methodology are the coverage criteria. We have defined two new criteria for framework-based product families, namely the hook and template coverages. They are based on traditional structural coverages and are simple to understand and use. As our example shows, they are useful in framework-based product family context. In our example, the hook and template coverages indicated results that would not have been indicated with currently used coverage criteria. We will continue our work to support these coverages in the existing RITA environment.

Acknowledgments
This work has been done as a part of the ITEA projects CAFÉ and Families. The authors would also like to thank Professor Jukka Paakki from the University of Helsinki.

References

1. van der Linden, F., Software Product Families in Europe: The Esaps & Café Projects. *IEEE Software*, Volume 19, Number 4, July/August 2002, 41–49.
2. Bosch, J., Product Line Architectures in Industry: A Case Study. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, California, USA, May 1999, 544-554.
3. Northrop, L. (director), *A Framework for Software Product Line Practice – Version 4.2*. Software Engineering Institute, Carnegie Mellon University, 2004, URL: http://www.sei.cmu.edu/plp/framework.html [May 27, 2004].
4. Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
5. Tevanlinna, A., Product Family Testing with RITA. *Proceedings of the Eleventh Nordic Workshop on Programming and Software Development Tools and Techniques (NW-PER'2004)*, Turku, Finland, August 2004, to appear.
6. Fayad, M., Schmidt, D., Johnson, R., *Building Application Frameworks*. Wiley and Sons, 1999.
7. Pree, W., Koskimies, K., Rearchitecting Legacy Systems – Concepts and Case Study. *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA'99)*, San Antonio, Texas, USA, February 1999, 51–64.
8. Alexander, R., Offutt A., Criteria for Testing Polymorphic Relationships. *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, San Jose, California, USA, October 2000, 15–23.
9. Chen, M-H., Kao, H., Testing Object-Oriented Programs – An Integrated Approach. *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE'99)*, Boca Raton, Florida, USA, November 1999, 73–82.