Reverse Generation and Refactoring of Fit Acceptance Tests for Legacy Code

The Fit framework is a well established tool for creating early and automated acceptance tests. Available Eclipse plug-ins like FITpro support for new requirements and new code writing of test data and creation of test stubs quite well. In our project we faced the problem, that a large legacy system should undergo a major refactoring. Before this, acceptance tests had to be added to the system to ensure equivalent program behavior before and after the changes. Writing acceptance tests manually for existing code is very laborious, cumbersome and very costly. However reverse generation of fit tests based on legacy code is not foreseen in the current Fit framework, and there are no other tools available to do so. So we decided to develop a tool which allows generation of the complete Fit test code and test specification based on existing code. The tool also includes automatic refactoring of test data when refactoring production code and vice versa, when changing the Fit test specification, it also updates production code accordingly. This reduces the maintenance effort of Fit tests in general and we hope, this will help to spread the usage of Fit for acceptance and integration testing even more.

Martin Kropp, Wolfgang Schwaiger | martin.kropp@fhnw.ch

For clarification we use the following terms for the rest of the proposal[1]. Production code is the code of the application, i.e. the source code of a system under test (SUT) and of a method under test (MUT), respectively. Test code in general reflects the code that is written to execute a test. Test data covers both input and expected data for a test. Moreover, a test case includes test data and the preconditions and expected postconditions. Finally, a test fixture comprises all that is needed to run a test, especially the test code and the test case.

Use Cases

When working with legacy code, a typical scenario might be that a developer has to add new functionality in a certain module. To be able do so, he may have to refactor certain areas in this module. Due to the often encountered lack of modularization in legacy systems and the missing tests, one approach is to add tests starting from the top level (i.e. on acceptance level). Having tests for the top level methods ensures that any misbehavior of the underlying refactored code will be detected immediately. The developer can use our iTDD (integrated Test Driven Development) tool to generate the test code and the related test data, based on the given method signature. The developer can then add new, or edit existing test data in Fit tables form within an IDE (e.g. Eclipse). The tests can be executed immediately and will indicate the success or failure of the test. In case of new code, the process is very similar: following the TDD approach, where the developer first specifies the signature of the new method, he can then generate Fit tests based on the new method. In this case, however, the test will fail as long as the implementation of the new method is not complete. Moreover, during maintenance as a developer refactors a method, the tool will automatically carry along the changes to the test data and the test code. So for a complete and integrated TDD approach we want to address all relevant activities in a software construction process [SWE], which result in the following use cases:

1. Generation of test fixture for legacy code.
2. Generation of test fixture for new code.
3. Refactoring of production code.
4. Refactoring of test code.
5. Refactoring of test cases.

3. Legacy Code Testing with Fit

Fit has been designed to enable early executable acceptance testing [Cun08]. The idea is, that starting from a use case specification, or from one scenario of a use case, the user can derive a concrete test case and specify it in tabular form. This tabular form represents a kind of view mock, which can then be implemented in code as a concrete Fit test. In this case the main purpose of the Fit tests are to validate and verify the user requirements. In the case of testing legacy code however, there is no need for requirements verification. In this case the main purpose of the Fit tests is to ensure the

1 The original version of this paper of the same authors will be presented at the OOPSLA conference. OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

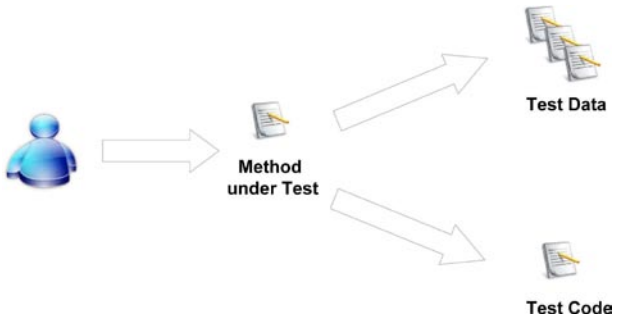

Figure 1: Separating test data from test code

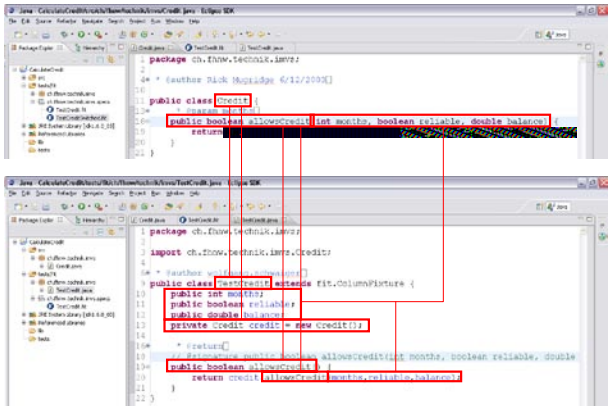

Figure 2: Mapping of production code and generated test code

equivalent behavior of the system on acceptance level when changing the underlying code base, i.e. providing a safe playground when changing the system.

To enable user centered approach for test specification the Fit framework provides a strict separation of test data and test code as indicated in Figure 1. Advantages of this concept are [MC05, Pet08]:

- Users can specify test cases in a user centered view, e.g. in HTML tables or even Excel spreadsheets.
- Test specifications can be defined by the end users and other stakeholders and not only by developers.
- There is only one test code implementation for all related test data of the same kind and thus does not increase with the number of specified test data.
- Test specifications do not change during code refactoring operations.
- Frameworks which use this separation enable better continuous integration of tests.
- Tests can be easily written on different levels of abstraction.

However, no lunch is for free, some drawbacks of this approach are [MC05, Pet08]:

- Connections between test data and test code have to be maintained outside of the test code.
- Developers have to use another, additional test framework.

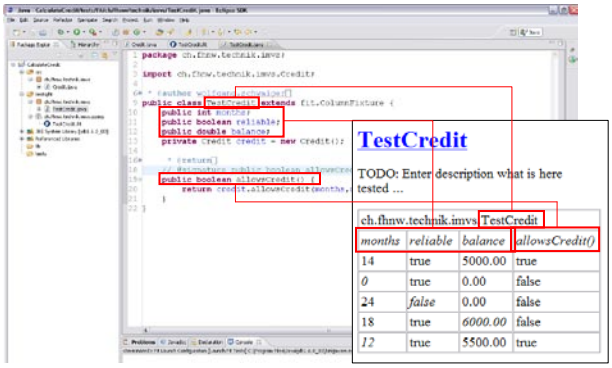

Figure 3: Mapping by convention strategy of test code and test data

Nonetheless, the advantages prevail, especially the significant test code reduction because of one common test code base for test cases of the same kind. This makes it easier to automate the test code generation. Also, the ability for the user to specify tests through the special test data viewer allows more profound test specifications.

iTDD Concepts

Fully automated test maintenance also includes the complete test code generation. Basically, the following two generation schemas can be differentiated.

- For new written applications (new code) the goal of a test suite is to map the functionality to the expected requirements. Usually, TDD starts with the implementation of a method stub – MUT – and then continuous with the specification of the expected behavior of the implementation in form of pre-written test cases. For this step, mainly the user jumps in (e.g. user stories) and leaves the developer with the implementation of the MUT. As mentioned before the tests will fail as long as the implementation is not complete.
- For legacy code the focus is more on the injection of tests into the system to assure not to break the functionality during refactorings, feature extensions, etc. Tests can be added by selecting the desired method in the SUT which is to be monitored and tested during change op-

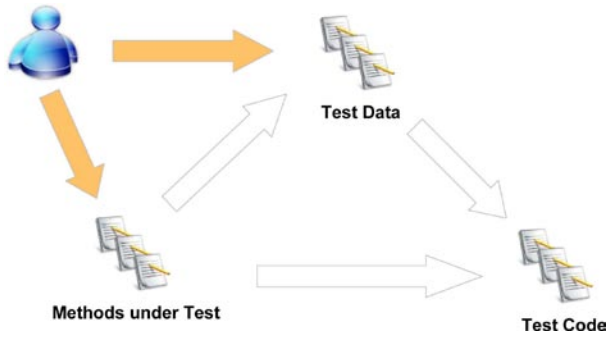

Figure 4: Tracing code changes

```
// production code
package math;
public class Calc {
   . . .
   public float divide (float n, float d) { return n/d ; }
}

// generated test code
package math;
public class CalcTest extends fit.ColumnFixture {
   public float n;
   public float d;
   private Calc calc = new Calc( );

   // test method
   public float d ivide ( ) {
      // call MUT
      return calc.divide ( n , d );
   }
}
```

Listing 1: Test code sample

erations. The tests are generated based on the given method signature and can be run immediately to prove the functionality.

For both cases the iTDD generates the complete Fit test code and the Fit HTML file containing the test specification as illustrated in Figure 3. The automated test generation process itself relies on a simple *mapping by convention strategy* which is used by Fit. This means, that, for example, the names of the column headers of the Fit test data tables correspond with the names of the parameters for the called method under test. The source information of the MUT is processed to extract all relevant information for automated test generation. Figures 2 and 3 illustrate the mapping of source code information with the generated test code and test data.

Listing 1 shows a very simple example of a method *divide* which takes two arguments and simply returns the quotient of *n* and *d*. In the lower part of Listing 1 the generated test code is stated which in this case extends the *ColumnFixture* fixture type. The test method has the same name as the MUT to enable a simple identification strategy for refactorings (see section Refactorings). Moreover, the declared field names are related to the class names and parameter names of the MUT.

Refactorings

While in general most attention is paid to the initial development of a software product, this phase makes only minor part of the overall cost of a software product. Actually, most effort on a software product is spent on the maintenance of the product through its whole product life cycle [Fea05]. Thus, any reduction in maintenance effort can bring a significant gain to reduce the overall development cost of a system. The iTDD tool addresses this issue by adding refactoring functionality also for Fit tests, i.e. for code as well as for test fixtures. The iTDD tool includes fully automated test code maintenance, making test code completely transparent for the developer. The developer and user can focus on changes that really matter to the system: improving the production code and improving test cases and test data. The test code is completely hidden and is adapted automatically through appropriate refactorings. The filled block arrows in Figure 4 indicate direct user interactions with the system, driven by external influences (e.g. requirements change). The white block arrows indicate adaptations which are carried along all impact files automatically by the tool. In software development changes may occur at various places:

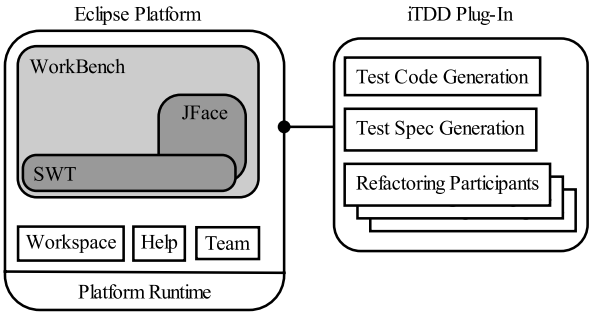

Figure 5: Architectural concept

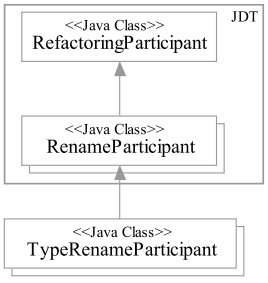

Figure 6: Extending Eclipse Refactoring

```
<extension point = ”org.eclipse.ltk.core.refactoring.rename Participants”>
<renameParticipant class =
    ”com.luxoft.fitpro.plugin.refactoring.participants.TypeRenameParticipant”
    id = ”itdd.participants.compilationunitrename”
    name = ”Compilation Unit Rename Refactoring Participant”>
  <enablement>
    <with variable = ”affectedNatures”>
      <iterate operator = ”or”>
         <equals value = ”org.eclipse.jdt.core.javanature”></equals>
      </iterate>
    </with>

  <with variable = ”element”>
     <instanceof value = ”org.eclipse.jdt.core.IType”></instanceof>
  </with>
  </enablement>
</renameParticipant>
. . .
```

Listing 2: Refactoring Configuration

1. a) The developer may change the production code due to refined requirements or to improve the code;
2. b) The user may also refine the test specification due to changed requirements or to improve readability;
3. c) The test engineer may change the test code implementation.

While a and b are mainly externally driven, namely by changing requirements, c is purely internal and usually seen as rather dry and time consuming. *Changing code* Existing toolkits (e.g JavaTM-Development Toolkit) support a wide range of code refactorings, so that they make a good foundation, on which the iTDD tool can build on. As shown in Figure 4 code changes do not only affect the production code but also the related test code and test data. The iTDD tool extends existing refactorings with additional components which propagate changes to the affected test files, e.g. column headers, class names, etc. (see section Eclipse Integration). *Changing test specification* As mentioned before the user may also change the test specification. He can enter new or modify and delete existing test data, for example. Except for the test data repository itself, this does not affect any other files. However, changes in the test specification may also concern changing of labels of input and output fields (e.g. to increase readability). These changes also have to be handled in the appropriate test code and might even propagate to the production code to keep naming and interfaces synchronized. Input fields might even be deleted or new fields be added due to changed requirements. This would finally lead to changes in the method signature of the production code. iTDD addresses this through the appropriate test specification refactoring which propagates the changes automatically to the production code.

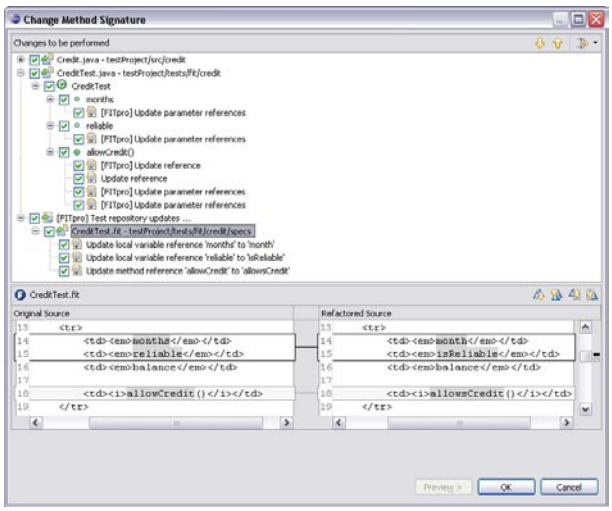

Figure 7: Refactoring UI Integration

Although the generation of test data was not in the focus of this work, the iTDD provides a simple random generator for test data. Moreover, the iTDD architecture provides a test data generator interface which allows easy integration of external test data generators. The interface allows specifying constraints for each individual parameter of a method, like max and min values, for example.

Eclipse Integration

In this chapter we will describe how we implemented the concept described above and how we integrated it into the Eclipse-IDE based on its plug-in concept and the Eclipse Abstract Syntax Tree (AST).

The iTDD tool has been implemented as an Eclipse plug-in [CR08] and consists of the two main modules for test code and test data generation, and various refactoring modules as depicted in Figure 5. The test code and data generation modules are actually integrated into one test generation module which appears as a new menu item in the Eclipse context menu. The refactoring modules are fully integrated into the Eclipse refactoring mechanism which is explained in the next paragraph.

Furthermore, to gather all relevant information for the automated test generation the Eclipse AST is used (e.g. to find class names, method calls, etc).

The Eclipse IDE already offers a wide variety of refactoring functionality in its Java Development Toolkit (JDT). The JDT also offers extension capabilities to add new refactorings for Java and for other file types and language types. The Eclipse refactoring architecture is based on the *Refactoring Participants* concept [Wid07]. This concept provides an extension mechanism, which is based on the idea to add new functionality as a new "participant" to any existing refactoring, like the Rename, Remove Parameter or Change Method Signature, for example, just as needed. To extend an existing refactoring with extra functionality, the new functionality must be implemented in a class derived from the corresponding base refactoring class as shown in Figure 6. As also indicated in Figure 6 multiple subclasses can be implemented for a given refactoring class to add a variety of new functionality to a given refactoring. This concept offers a very modular and flexible extension of existing Eclipse refactorings.

Additionally all new refactoring participants must be registered as extension points in the *plugin.xml* configuration as shown in Listing 2. So far we have implemented refactoring extensions for *Class Rename*, *Method Rename*, and *Change Method Signature* to include the refactoring of the corresponding Fit test specifications. The new refactoring extensions are fully integrated into the standard Eclipse refactoring process and the new functionality appears in the standard refactoring dialogs as shown in Figure 7.

Conclusion

The current implementation of iTDD includes the complete generation of test code and test specification for a given method signature. The automated generation is currently supporting functional tests (i.e. fit ColumnFixtures), which presumes a signature of the method under test to have input parameters and a return value. The generation process supports all basic data types as well as custom data types. For test data generation we currently use a simple random data generator for the basic data types. When generating tests for multiple methods of the same class the tool adopts the existing test code and test specification files accordingly. Refactoring operations for renaming classes, methods, and parameters are fully integrated by extending the refactoring tooling of the Eclipse IDE. Moreover, the change method signature and the modification of test specification operations are fully integrated. We are currently integrating the refactoring functionality into the open source tool FITpro [Lux] and will hopefully submit it to the community by the end of the year. Future work will focus on the extension of the iTDD tool and the operative application. Enhancements of the tool will include full support for custom data types which includes the generation of the appropriate test data. We will also implement test generation for overloaded methods and constructors. Strategies for test generation from input dialogs and workflows will be addressed in the future (see ActionFixtures and RowFixtures [MC05]). Further work could cover the development and integration of test data generation based on contract information such as given by Contract4J [ARA] to improve the quality of the test data

Acknowledgments

The authors give many thanks to the Hasler Foundation who funded this work as part of the ProMedServices project.

References

[ARA] Aspect Research Associates. Contract4J. http://www.contract4j.org (09.10.2008), 2003–2008.

[CR08] E. Clayberg and D. Rubel. Eclipse Plug-ins. Addison-Wesley Professional, Reading, Massachusetts, third edition, December 2008.

[Cun08] W. Cunningham. Framework for Integrated Test – Fit. http://fit.c2.com/ (10.10.2008), 2008.

[Fea05] M. C. Feathers. Working Effectively with legacy Code. Pearson Education, Inc., New Jersey, 2005.

[Lux] Luxoft UK Limited. FITpro – Acceptance Testing Solution. http://sourceforge.net/projects/fitpro (02.10.2008), 2008.

[MC05] R. Mugridge and W. Cunningham. Fit for Developing Software. Pearson Education, Inc., New Jersey, 2005.

[Pet08] D. Peterson. Concordion. http://www.concordion.org (01.10.2008), 2007–2008.

[PMS08] Hasler Foundation. ProMedServices – Proactive Software Service Improvement. http://web.fhnw.ch/technik/ projekte/promedservices (19.03.2009), 2008.

[SWE] IEEE Comnputer Society. SWEBOK. http://www.swebok.org/ (27.01.2007), 2007–2008.

[Wid07] T. Widmer. Unleashing the Power of Refactoring. IBM Rational Research Lab Zurich, February 2007.