Support for DbUnit in NetBeans IDE

Bachelor’s Thesis

Pavol Pitoňák

Brno, May 2009
Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: Ing. Petr Adámek
Abstract

This thesis is focused on creating a module that would integrate DbUnit into NetBeans IDE. The introduction to DbUnit explains why unit testing is important and also explains how to make testing of database applications much easier. Brief introduction to the NetBeans Platform helps to understand the Platform, its APIs, and shows how it can help programmers to save their time and resources. In the following chapters, the desired features of the module are described and the process of creating the module is explained.
Keywords
Java, NetBeans Platform, NetBeans IDE, DbUnit, testing, database, module
## Contents

1 Introduction .................................................. 1
2 DbUnit ....................................................... 3
  2.1 Unit Testing .................................................. 3
  2.2 Testing of Database Applications .......................... 3
  2.3 Testing Using DbUnit ........................................ 4
  2.4 Data Sets .................................................... 4
  2.5 Assertion ..................................................... 5
3 The NetBeans Platform ......................................... 6
  3.1 History ....................................................... 6
  3.2 Modular System ............................................. 6
  3.3 The System FileSystem ..................................... 7
  3.4 DataObjects, Nodes and Explorer .......................... 7
  3.5 Window System ............................................. 7
  3.6 Other APIs ................................................ 8
4 The DbUnit Module .............................................. 9
  4.1 About the Module ........................................... 9
  4.2 Wizards .................................................... 10
  4.3 Internationalization ...................................... 10
  4.4 JavaHelp .................................................. 11
  4.5 Using Libraries ........................................... 11
5 Generating a Single File ....................................... 13
  5.1 Generating Files ............................................ 13
  5.2 Templates and FreeMarker .................................. 13
  5.3 Empty Test Wizard ......................................... 13
  5.4 Simple Test Wizard .......................................... 14
  5.5 Two Ways of File Generating ............................... 14
  5.6 The Java Source and Javac API ............................ 16
  5.7 Registering Templates ..................................... 17
6 Generating Multiple Files ..................................... 19
  6.1 Context Menu and Action ................................... 19
  6.2 Enabling the Action ........................................ 19
  6.3 Performing the Action ....................................... 20
  6.4 Registering the CreateTestAction .......................... 20
7 DataSet Generator ............................................... 23
7.1 Creating DataSets .................................................. 23
7.2 The Wizard .......................................................... 23
8 Conclusion .............................................................. 24
Bibliography ............................................................. 25
A Data Sets ............................................................... 26
B FreeMarker Template ............................................... 27
C Contents of Attached CD ............................................ 29
Chapter 1

Introduction

In the last decade, information technologies have become an important part of everyday life. Unlike programs in 1960s, contemporary programs are much bigger and more complicated. Not only do users require a lot of functionality from their programs, but they also require high quality. There are many points of view on the quality of software. For programmers, the most important feature of software is its correctness, meaning that it does what it should do. Unit testing is one of the many techniques for decreasing the number of bugs in a program early in the development cycle.

Unit testing is a method that verifies both that the units of source code meet software requirements and that it behaves as the developer intended. In object-oriented programming, the smallest unit is a method. Since units have to be tested in isolation, one needs to use stubs, mocks, or fake objects. Unit testing in Java can be simplified by using a framework such as JUnit or TestNG.

The second chapter of this thesis will discuss DbUnit, a JUnit extension that simplifies writing unit tests for database-driven projects that, among other things, puts the database into a known state between test runs. DbUnit has the ability to export and import the database data to and from XML data sets. Since version 2.0, DbUnit can also work with very large data sets when used in streaming mode. DbUnit can also help to verify that the database data match an expected set of values [1].

NetBeans IDE, one of the most popular integrated development environments (IDE), will be discussed in the third chapter. Given the context of this project, NetBeans APIs are covered more closely. The NetBeans Platform is a modular and extensible application framework for the development of Java desktop applications. It is the core of the NetBeans IDE, and the NetBeans IDE serves as an SDK for the Platform. The NetBeans Platform contains several APIs, thus providing the functionality that every programmer would have to write themselves, including: window management, saving state, connecting actions to menu items, toolbar items and keyboard shortcuts, etc. It allows the developer to focus on the problem domain and save a huge amount of time and work.

There are plenty of modules available for NetBeans IDE that make the development of Java programs easier. There is also a module that integrates JUnit into NetBeans IDE. With this module, one can generate the skeleton of a unit test not only for one’s classes but also for whole packages using wizards. Test can be run from the main menu using a shortcut or from context menu. The results of tests are shown in a new window where the developer sees which tests failed and which passed. After clicking on a test name, the developer can see
the reason for failure. It is also possible to launch only failed tests. There is no module that
would simplify the use of DbUnit in NetBeans IDE in the same way that the JUnit module
does for JUnit.

In the fourth chapter, the basic concepts of the DbUnit module are introduced. The follow-
ing two chapters describe all of the important parts of the module. There are two ap-
proaches explained for generating files, and the Java Source and Javac API are mentioned.
The creation of context-aware actions is also explained in these chapters. In the last chapter,
a wizard for the data set generators is described.
Chapter 2

DbUnit

2.1 Unit Testing

The goal of unit testing is to isolate each part of the program and show that the individual parts behave as expected. Each unit is tested separately for programmer to make sure that a bug from another unit will not show up in the tested unit. A unit test provides a strict written contract that the piece of code must satisfy. Many unit testing frameworks, which help simplify the process of unit testing, have been created, among which JUnit and TestNG are the most popular.

JUnit is a simple open source framework designed to write and run repeatable tests based on a design by Kent Beck [2]. JUnit includes assertions for testing expected results and test runners for running tests. The environment has to be in known state (called the test fixture) when the test is started. Writing shared fixture code can save a lot of time because it is often possible to use the same fixture for several different tests. All the programmer has to do is annotate the method with @BeforeClass or with @Before annotations.

2.2 Testing of Database Applications

There are two main approaches to testing database applications. The true unit test should not work with the database. This may be done using a mock object. In object-oriented programming, mock objects are simulated objects that mimic the behavior of real objects in controlled ways. A computer programmer typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in vehicle impacts [3]. In this case, the mock object would simulate an RDBMS. The testing purists argue that this is the right approach but, on the other hand, there is no better way to verify data mapping than testing against a real database. The one simple workaround might be to use an H2 Database Engine. This is an advanced, open source, in-memory Java database with a small footprint.

The following paragraph is based on Richard Dallaway’s article [4]. The usual database development consists of setting up a database, writing code which accesses that database, running the application, and, in the end, checking the database to verify that there are all records. There are many problems with visual inspection that make such a test inefficient.

---

1. Relational Database Management System
These problems can be solved by using a framework that automates the whole process of testing. Automated tests, i.e. tests that run often and test a lot, reduce the chances that data will be missing.

2.3 Testing Using DbUnit

The DbUnit is a database testing framework that extends JUnit and sets up the database in a known state before executing the tests. This framework uses XML data sets and performs database operations before and after each test.

There are some simple rules that must be obeyed by the developers [1]. Testing can be simplified if the database is in a known state before running the test. A database should only be used for one test at a time, otherwise the database state can not be guaranteed. Multiple developers working on the same project should have their own local databases to prevent data corruption and to make sure that the database is in a known state. The last rule says that one should always avoid creating tests that depend on results of preceding tests—this is why the DbUnit was created.

Most of tests do not require the entire database to be re-initialized. A good practice is not to create one large data set but to break it into several smaller data sets. This will help to reduce overhead caused by initializing the database for each test. When needed, it is still possible to use the CompositeDataSet class to combine multiple data sets into a large one at run time.

2.4 Data Sets

The IDataSet interface represents a collection of tables. This is the primary abstraction used by DbUnit to manipulate tabular data. It has about 20 implementations, from which the most commonly used are [1]:

- FlatXMLDataSet—a data set where each XML element name corresponds to a table name and the XML attributes (there is an example in Appendix A);
- XmlDataSet—a more verbose version of the FlatXMLDataSet;
- StreamingDataSet—a data set providing cursor-like forward-only access to data and only keeps the active row in memory;
- QueryDataSet—a data set holding tables resulting from database queries;
- CompositeDataSet—combines multiple data sets into a single logical data set; and
- XlsDataSet—a data set working with MS Excel spreadsheets; each sheet represents a table, the first row of a sheet defines the names of columns and remaining rows contain the data.
In the DbUnit module, the generated tests work with FlatXMLDataSets. The data set generator (chapter 7) uses the QueryDataSet so that it is easy to limit the size of the data set using SQL.

2.5 Assertion

The DbUnit framework comes with its own class Assertion providing static methods for the most common DbUnit assertion needs. There are methods for comparing two data sets, two tables, a table and the result of a database query, or a data set and the result of a database query.
Chapter 3

The NetBeans Platform

3.1 History

The NetBeans IDE (originally called Xelfi) started as a student project in 1997 at Charles University in Prague—it was the first Java IDE written in Java. Shortly thereafter, a company was formed to sell the commercial version of the NetBeans IDE. In 1999, the company was bought by Sun Microsystems and open sourced in June 2000. In October 2007, Sun announced that NetBeans would be offered under a dual license of the Sun’s CDDL\(^1\) and the GPL version 2 with classpath exception. See [5] for the whole history of NetBeans.

3.2 Modular System

Applications are getting more complex, they are assembled from many pieces, and they are being developed by teams distributed all over the world. For these reasons, it is very hard to manage the development. One possible solution is to design a modular architecture for the software. This can be done by separating the API from the implementation. This approach allows the programmers to make multiple implementations, but the API needs to be able to find all its implementations at runtime.

A modular application is able to add, remove, and reload components at runtime. These operations are performed by a runtime container. The runtime container in the NetBeans Platform consists of five modules: Bootstrap, Startup, Module System API, FileSystem API, and Utilities API. The module system must satisfy dependencies between components. It can not allow illegal dependencies; it only allows legal dependencies and provides service registration and a discovery facility. Lookup, NetBean’s extension of the Java Extension Mechanism, is a small NetBeans independent library that provides all of the above-mentioned functionalities, and unlike JDK’s ServiceLoader class, Lookup works with any version of Java.\(^2\)

A NetBeans module is a JAR with some specific metadata in its Manifest file. There are specified interfaces that it provides and classes that implement public interfaces declared by other modules. There are also defined dependencies and a layer.xml file that will be discussed later. In NetBeans IDE, there is an option to create an NBM file. This type of archive is usually created for the module to be distributed easily.

---

2. The ServiceLoader class was added in Java 6.
3.3 The System FileSystem

The FileSystem is a general registry of configuration data. It is a virtual filesystem that works with FileObjects, not java.io.Files. Unlike java.io.File, a FileObject represents an existing file in the filesystem, and it is not usual to create FileObjects. The main advantages of a FileObject are: it has MIME type, attributes, and input and output streams. Another significant feature is that one can listen for changes in FileObjects [8]. In NetBeans, FileSystems are the reason that it is possible to expand a JAR file and open files within that JAR file just as if they were plain files on disk. It is completely transparent to one’s code whether the file is on the local disk, in a JAR, or on an FTP server [7, p. 70].

There is a layer in every module—in the JAR it is represented by layer.xml file. These layer.xml files are not only in module JARs, but also in NetBeans’ installation directory and user directory. Each layer contains declarations for actions, editor configurations, Options window settings, window persistence, libraries, and much more. All of these layers together create the System FileSystem. The layer.xml has to be registered in the module’s Manifest:

```
1  OpenIDE-Module-Layer: cz/muni/fi/dbunit/resources/layer.xml
```

3.4 DataObjects, Nodes and Explorer

A DataObject is a logical object around a FileObject. It adds to FileObject more than just plain content. It gives each file a logical behavior, i.e. an icon, a name, operations, etc. Multiple files can be represented by one DataObject. In the NetBeans IDE, this is used, for example for resource bundles. There is one logical object per resource bundle but it represents more physical files in a filesystem. Another example is Java sources—there is one object for each Java source and class file.

A node provides a visual representation of most objects in NetBeans. Nodes are a variant of JavaBeans that may have adjustable property sets, provide cookies, and more. It may be used to represent a DataObject from the Datasystems API. Using the Explorer API, one can create a variety of different presentations for any tree or subtree of nodes hierarchy. The Explorer API provides UI components to render nodes in trees, lists, combo boxes, and tree tables [6].

3.5 Window System

The most visible part of the NetBeans is its Window System. The window manager is a system for managing all of NetBeans’ windows. When the developer wants to create a new window, the TopComponent class has to be subclassed. Using the window manager, one is able to move windows, dock and undock them, make them transparent, etc. It is possible to define group behaviors for windows. There is also an experimental support for special effects. The Matisse GUI Builder can be used for designing new TopComponents. Persistence of windows is one of the most useful features of the NetBeans’ window system. The last
state of TopComponents is automatically persisted when the application is quit. Of course, the programmer has the option to modify the default persistence code.

The window manager provides a wide functionality, but this is not always what is needed. It is very easy to limit window system’s behavior and functionality. For example, it is possible to disable moving, undocking, and closing windows. Though it is easy to make changes for the whole window system, until NetBeans 6.7 (June 2009), it was not easy to make changes for a specific window. Another limitation of the NetBeans’ window system is that it is not easy to replace NetBeans’ window system with another window system.

3.6 Other APIs

The NetBeans Platform contains a lot of other APIs that can help with many tasks. There is an API for automatic updates, creating editors with code completion, code folding and syntax highlighting, integrating JavaHelp into an application (section 4.4), writing lexers for new languages, for printing, and many more. It contains a Settings API that makes the persisting of user settings much easier. Some APIs that are related to Java projects will be discussed in the following chapters.
Chapter 4

The DbUnit Module

4.1 About the Module

The whole project was created as a suite of modules containing three modules. There is a core module and two modules that wrap libraries needed by the core module, namely one for DbUnit and one wrapping Simple Logging Facade for Java (SLF4J). The SLF4J library is one of DbUnit’s dependencies and it is the only dependency needed for DbUnit to run in its basic configuration. The complete list of all DbUnit’s dependencies can be found on the DbUnit home page [1]. Library wrapper modules are very simple modules that contain only library JARs and a resource bundle. These modules are loaded automatically when the core module is installed.

The core module is divided into three packages of which none are public because there are only implementation classes. There was no need to create a public API because there is only one implementation and no other implementations made sense. The module was created with an emphasis on maximal reusability. Since the DbUnit is an extension of JUnit, the DbUnit module should extend the JUnit module’s functionality and thus it should reuse as much code as possible. The scale of reusability will be described in subsequent chapters.

The module consists of four logical parts. The first is an action (section 6.1) visible in a popup menu and a main menu that creates tests for existing classes, the others are wizards, available from the New File menu, that serve as generators of:

• empty DbUnit tests;

• tests for existing classes; and

• classes intended for creating DataSets.

In the NetBeans IDE, the convention is to put the layer.xml file to the resources, so it was moved there. This change implied the need of another change—in the module’s Manifest. The resources package also contains an icon, file templates, template descriptions, and the DbUnit library descriptor.

---

1. [http://www.slf4j.org](http://www.slf4j.org)
4.2 Wizards

In the Module Development group of the New File wizard, there are many wizards that do the tedious part of the work. There is even a wizard for creating wizards. This wizard asks for its type, number of steps, class name prefix, display name, category, and package. Then it generates a template HTML descriptor, a visual panel, a wizard panel, and a wizard iterator. The wizard also registers the template in the layer.xml and adds needed strings to the Bundle.

A wizard iterator is a class implementing the interface InstantiatingIterator. It is a controller of the wizard that has two noteworthy methods. The first is a private method getPanels that is called from public methods, and which instantiates the panels of the wizard. Using this approach it is possible to instantiate the panels as late as possible. The second important method is the public method instantiate that is called after the Finish button is pressed. In this case, it means that the generation of files is executed from here. The rest of the methods serve as support methods for navigating in wizard.

A wizard panel is an implementation of the WizardDescriptor.Panel interface. It is a wrapper class for a visual panel, that is an ordinary JPanel. The wizard panel contains the method getComponent that instantiates a new visual panel. Similar to the wizard iterator, it is also designed in such a way that the component is not instantiated until needed. More interesting are the methods storeSettings and readSettings that are used for panel persistence.

```java
public void readSettings(Object settings) {
    Preferences pref = NbPreferences.forModule(EmptyTestWizardPanel.class);
    component.setSourceCodeHints(pref.getBoolean("etw_sourceCodeHints",true));
}
```

```java
public void storeSettings(Object settings) {
    Preferences pref = NbPreferences.forModule(EmptyTestWizardPanel.class);
    pref.putBoolean("etw_sourceCodeHints", component.isSourceCodeHints());
}
```

It is possible to store booleans and other primitive types, Strings, and arrays of bytes.

4.3 Internationalization

Both the NetBeans Platform and the NetBeans IDE are internationalized very well and are available in several languages. All of the core modules of the NetBeans IDE are internationalized and thus the DbUnit module should be as well. All GUI components were designed using Matisse GUI Builder, which has very good support for internationalization. In fact, for all strings that appear in GUI, the Matisse GUI Builder inserts keys into the resource bundle and modifies the source code to use the class NbBundle discussed below.

In a plain Swing application, the ResourceBundle class is used, but in a NetBeans-based application the NbBundle class is used. NbBundle is similar to ResourceBundle

4. THE DbUNIT MODULE

but has some extra features: it provides better caching mechanism and works with a branding infrastructure, so strings retrieved from the NbBundle can be substituted. This is why a NetBeans-based application does not show up with the word “NetBeans” in its window title and other parts of its UI. The Class parameter is used to discover the package in which to look for a .properties file. NbBundle always looks for a file called Bundle.properties in the same package as the Class object passed to it [7, p. 569-570].

```java
package com.mycom;
public class Foo {
    private static String theMessage = NbBundle.getMessage (Foo.class, "Foo_theMessage");
}
```

In the code snippet above, the NbBundle searches for the key “Foo_theMessage” in the file /com/mycom/Bundle.properties and returns the string belonging to this key. Depending on user’s localization settings, it might also look in Bundle_sk.properties, etc.

In a similar way, it is possible to internationalize mnemonics simply by placing an ampersand character before the character in the string that should be the mnemonic. There is the Utilities.loadImage method for loading PNG, JPG and GIF images. It works with the NetBeans module classloaders and the branding mechanism [7, p. 571]. This can be used for loading localized images.

4.4 JavaHelp

The JavaHelp system is an open source project that can add a built-in set of HTML help topics to the NetBeans Platform application. Each NetBeans module can provide its own JavaHelp help sets [7, p. 441]. The IDE merges the individual help sets into a single whole. In the NetBeans IDE, there is a wizard for creating the help set. This includes creating a help set descriptor, index, table of contents, and the mapping of keys to files with topics. It also creates a simple topic and modifies the layer.xml and Manifest. It registers the help set in the layer.xml like this:

```xml
<folder name="Services">
    <folder name="JavaHelp">
        <file name="dbunit-helpset.xml" url="dbunit-helpset.xml">
            <attr name="position" intvalue="3211"/>
        </file>
    </folder>
</folder>
```

4.5 Using Libraries

Since the module uses the DbUnit library and the DbUnit has some dependencies, there should be a new library registered in the Library Manager and, after the first test is created, the library should be added to the tests' classpath. In the NetBeans IDE, there is a wizard for creating library descriptors using libraries registered in the development IDE. Though it
is a useful feature, it is not usable for the DbUnit module because it does not generate links to other modules. Registering the library descriptor in the layer.xml is very simple:

```xml
<folder name="org-netbeans-api-project-libraries">
  <folder name="Libraries">
    <file name="DbUnit.xml" url="DbUnit.xml"/>
  </folder>
</folder>
```
Chapter 5

Generating a Single File

5.1 Generating Files

The main objective of the DbUnit module is to generate files with Java classes. There are many ways to generate such files. The simplest way is to use the package java.io and its system of output streams and writers. Though this might be the easiest approach it is hardly reusable and extensible, if at all. The most straightforward way is to generate files using templates. This approach is used for generating almost all types of files in NetBeans IDE. All templates can be previewed and modified in the Templates menu. It is used also by the JUnit module, so the DbUnit module will use templates for all tests, either directly or indirectly (this will be explained later in this chapter).

5.2 Templates and FreeMarker

The user usually creates tests using the New File wizard. The user chooses the file name and then presses the Finish button. Then the file is created from the template. In NetBeans, a file template consists of a template file, an HTML description file, and an icon. The HTML description file displays information about the template in the New File wizard. The icon helps to distinguish the template from the other templates in the New File wizard [9].

The developer can choose whether he or she will use a plain text template or will use a template engine. Almost all of the templates in NetBeans IDE use FreeMarker—an open source template engine. This is a generic tool used to generate text output. FreeMarker is designed to be practical for the generation of HTML web pages, particularly by servlet-based applications following the Model View Controller pattern [10]. Since the FreeMarker is generic, it is possible to use it not only for generating HTML files but also for other types of files, such as Java classes. There is an example template in Appendix B: it is a template used to generate an empty JUnit 4 test.

5.3 Empty Test Wizard

The first implemented part of the DbUnit module was the wizard for creating empty DbUnit tests. It is a standard wizard with two static panels. The first panel consists of two parts: the top part is a package chooser from the module Java Project Support and the bottom part is
its own wizard panel, EmptyTestWizardPanel. The package chooser is a standard component used for creating all kinds of files in Java projects. It asks for the file's name, location (e.g., Source Packages), and package. In the bottom, there is an EmptyTestWizardPanel that contains three checkboxes for usual JUnit settings.

The next step of the wizard is a panel with the DbUnit's settings. The user can set the database connection, choose a data set, and choose whether the module will generate DbUnit-specific methods or not. This panel was designed in such a way that it can be reused by all wizards in the module. The database connections are loaded into a combobox using a utility method:

```java
DatabaseExplorerUIs.connect(connectionComboBox, ConnectionManager.getDefault());
```

After clicking on theBrowsebutton, a new DataSetChooser is instantiated and then, using Dialogs API, a new dialog descriptor is created and the dialog is shown.

```java
DataSetChooser myChooser = new DataSetChooser(node);
final DialogDescriptor dd = new DialogDescriptor(myChooser,
    NbBundle.getMessage(DbUnitVisualPanel.class, "DbUnitVisualPanel.windowTitle"));
...
Dialog dlg = DialogDisplayer.getDefault().createDialog(dd);
dlg.setVisible(true);
```

### 5.4 Simple Test Wizard

The wizard for creating tests for a single class is very similar to the previous wizard. The first panel could not be reused from the module Java Project Support because there was a need for completely different components, so the SimpleTestWizardPanel was implemented. The main part of this panel is the class chooser that appears when the user clicks on theBrowsebutton. There are also many more options for generating tests, all of which are options for JUnit. The second panel is the same panel that was used in the first wizard.

### 5.5 Two Ways of File Generating

As was mentioned earlier, all files are generated using templates, either directly or indirectly. The direct way is used for generating empty tests and data set generators (chapter 7).

In the following code snippet, there is a part of the iterator's instantiate method that creates the file from the template. At first, a map containing arguments to be passed to the FreeMarker has to be created (line 1). In the next step, a FileObject representing the template is obtained from the wizard descriptor, and it is converted to a DataObject.
Generating a Single File

(line 4) using which the test will be generated. Then, using the utility class Templates, the target folder is obtained from the wizard descriptor and converted to a DataFolder (line 7). Subsequently, the class’s name is gained using the Templates’ getTargetName method and this is used to create an empty DbUnit test (line 11). Finally, the created DataObject is converted to a FileObject so that it can be returned from method instantiate. This method, if found, adds the necessary JARs to the test’s classpath. In the end, it returns the set containing the created file.

```java
Map<String, Object> args = prepareArgsForFreeMarker();

FileObject template = Templates.getTemplate(wizard);
DataObject dTemplate = DataObject.find(template);

FileObject dir = Templates.getTargetFolder(wizard);
DataFolder df = DataFolder.findFolder(dir);

String targetName = Templates.getTargetName(wizard);
DataObject dobj =
  dTemplate.createFromTemplate(df, targetName, args);

FileObject createdFile = dobj.getPrimaryFile();
```

The wizard for creating a test for a single class is very similar, but there is a big difference in the instantiate method. In this case, the file creating process is divided into two steps: creating an ordinary JUnit test and adding methods specific to DbUnit. Creating the JUnit test is done using this code:

```java
JUnitPlugin plugin = TestUtil.getPluginForProject(Templates.getProject(wizard));
final FileObject[] testFileObjects =
  JUnitPluginTrampoline.DEFAULT.createTests{
    plugin,
    new FileObject[]{classToTest},
    testDir,
    params};
```

The class JUnitPlugin is a Service Provider Interface (SPI) for custom implementations of support for JUnit. It declares methods for the creation of a test class skeleton and for navigating between source classes and corresponding test classes. Since in the DbUnit module the standard plugin from the JUnit module was used, there is no need to implement these methods, as JUnit’s behavior is convenient for the DbUnit module. The params variable is a map with keys of type CreateTestParam (a class from the JUnit module), and with values of type Object. It contains settings for JUnit’s test generator, similar to the map passed to the FreeMarker mentioned earlier in this section. Internally, the JUnit module uses templates to create tests. So the DbUnit module uses templates too, but indirectly.
5. Generating a Single File

5.6 The Java Source and Javac API

From the DbUnit module’s point of view, the second step of creating a test is more interesting. Based on the user’s settings from the wizard’s panels, the DbUnit module has to add DbUnit-specific methods, e.g. `getDataSet` or `getConnection`, an extends clause, and import clauses for all used classes. A combination of the Java Source and the Javac API modules is used for all of these operations. The main class of this module is `JavaSource`, which represents a parsed source file. The use of this class might look like this:

```
FileObject classFile = ...;
JavaSource classSource = JavaSource.forFileObject(classFile);
classSource.runModificationTask(new FooTask()).commit();
```

`JavaSource.runModificationTask` method passes the working copy to a task, i.e. the class implementing the interface `Task`. The task has only one method—`run`—that performs the source code modifications. It uses the `Javac API Wrapper` module in which the interface `Tree` can be found. The interface `Tree` is the main type used for manipulating the source files. It has many implementations for various parts of a source file, from which the below were used in the DbUnit module:

- `ClassTree`—a tree node for a class, interface, enum, or annotation type declaration;
- `AnnotationTree`—a tree node for an annotation;
- `CompilationUnitTree`—represents the abstract syntax tree for compilation units (source files) and package declarations (package-info.java);
- `ModifiersTree`—a tree node for the modifiers, including annotations, for a declaration;
- `VariableTree`—a tree node for a variable declaration;
- `ExpressionTree`—a tree node used as the base class for the different types of expressions; and
- `MethodTree`—a tree node for a method or annotation type element declaration.

In the DbUnit module, there are eight classes implementing the interface `Task`: a task for adding the extends clause, a task for adding import statements, and six tasks for DbUnit-specific methods. Here is the method `run` of `AddImportTask`:

```
@Override
public void run(WorkingCopy workingCopy) throws Exception {
    workingCopy.toPhase(Phase.RESOLVED);
    CompilationUnitTree cut = workingCopy.getCompilationUnit();
    TreeMaker make = workingCopy.getTreeMaker();
    CompilationUnitTree copy = make.addCompUnitImport(
```
At first, a CompilationUnitTree is obtained from a working copy (line 5). As mentioned earlier, a CompilationUnitTree represents the whole source file. In the next step, an instance of the TreeMaker is obtained using the WorkingCopy’s getTreeMaker method. The TreeMaker is a factory class for creating new Tree instances. It defines methods for creating trees for all kinds of clauses and statements. In the previous example, there was a new ImportTree created.

5.7 Registering Templates

Once the file template is created, it has to be registered in the System FileSystem that was explained in section 3.3. The wizard for creating wizards registers the template in the layer.xml, but this process is not sufficient for the DbUnit module.

There is a file defined within the Templates folder in category JUnit. The file’s url attribute is set to the location of the template. There is specified the position of this template in the New File dialog—it will be displayed after JUnit’s templates. If the template attribute was not true, the NetBeans IDE would not display the template in the New File dialog. The attribute templateWizardURL is a link to the HTML description that is also displayed in New File dialog. The instantiatingIterator defines the iterator that will be used to manage the wizard. There, the resource bundle used for localization is also set. In the Bundle there should be a line like:

```
Templates/JUnit/EmptyDbUnitTest.java=DbUnit Test
```
5. Generating a Single File

This string together with HTML description and the icon makes the use of the template more user-friendly. The last attribute is used to define the template engine. Since all of DbUnit’s templates use FreeMarker for generating files, this attribute is compulsory. Otherwise, the template mechanism would take the template as is and FreeMarker’s markups would appear in the file.
Chapter 6
Generating Multiple Files

6.1 Context Menu and Action

Another way to create tests is by clicking on an item in the popup menu of a node representing source file. In this way, it is possible to create tests for all selected files. All menu items are implemented as Swing’s actions. An abstract class `SystemAction`, which is part of the `Utilities API`, implements Swing’s `Action` interface. There are two subclasses of the class `SystemAction`, whereas the `CallableSystemAction` is the more frequently used one. Actions subclassing the `CallableSystemAction` are enabled all the time, but what is suitable for most of the use cases is not suitable for the DbUnit module. DbUnit’s action has to be enabled if and only if the Java source files from one project are selected. It can not even be enabled for Java source files stored in the test directory. This is the reason why an action implementing the interface `ContextAwareAction` should be used. Since the action will be in the nodes’ popup menu and it has to be context-aware, one solution is to extend an abstract class `NodeAction`. Another possible way would be to extend a `CookieAction`, but this approach would be much more complicated.

When creating a new custom wizard from the `New File` menu, visual panels, wizard panels wrapping visual panels, and an action are all created. This action extends the NetBean’s `CallableSystemAction` (whose disadvantages were explained earlier in this section). In the DbUnit module, there is a `CreateTestAction` that extends the `NodeAction`. It was created using the above-mentioned wizard and then the extends clause was rewritten. This caused the method `performAction()` to be redundant but, on the other hand, the methods `performAction(Node[])` and `enable(Node[])` must had to be added.

The `CreateTestAction` controls a two-panel wizard where the first panel is a panel used for generating tests for a single class (`SimpleTestWizardPanel`) and the second is the `DbUnitWizardPanel`. This reusing of existing panels saves a lot of time, and so only the two new methods had to be implemented.

6.2 Enabling the Action

The first important method in the `CreateTestAction` is the method `enable`. The array of `Nodes` passed to this method contains nodes that represent all files selected by a user in the `Projects` window. As mentioned in previous section, the action can not be enabled all the time while the NetBeans IDE is running. The method `enable` is called each time the user’s
file selection changes, and considering the method’s return value the action is enabled or disabled. The method was implemented in the same way as the JUnit module’s method `enable`. Reusing support classes from the JUnit module speeds up the development of an action.

In most cases, there is just one node selected, so this case is handled in a special, more effective way. There are no collections and iterators used. First, it needs to check that a `DataLoader` for the selected file exists so that it is possible to open the file and convert it into a `DataObject`. Then it checks that the file is valid. It may become invalid if it was deserialized and the file no longer exists on disk, or if the file was deleted. Continuing, it checks that the file is part of a project and that the file’s extension is “java”. If all of these tests are passed, the JUnitPlugin’s `canCreateTests` method finds out whether the file with the Java class is testable. It is not testable, e.g., if it has no public, protected, and package-private methods.

In principle, for additional selected files, the method is implemented in the same way. It has to perform the same validations, but there is a need to perform them in a for-each loop.

### 6.3 Performing the Action

The most important part of the CreateTestAction is the method `performAction`. It has one parameter—an array of `Nodes`. These nodes represent files from the `Projects` window that the user has selected. It is divided into two parts. In the first one, a new wizard descriptor is created and is then used to create a new dialog. Subsequently, the dialog is made visible and shown in front of any other window (so called focused window). In the second part of the method `performAction`, the return value of the wizard is checked, and, in the case that the user clicked the `Finish` button, the private method `instantiate` is called. Until now, it was not necessary to perform file generation in a new thread because it was done automatically.

The method `instantiate` is very similar to the method `instantiate` from the iterator of the wizard generating test for a single class. At first, it has to create ordinary JUnit tests using classes from the JUnit module. Then, a method adding DbUnit-specific methods, import statements, and an `extends` clause is called for each file. This method is called from a new thread so that the GUI does not freeze and so that finished files can be opened. Again, the file modifications are done using the `Java Source` and the `Javac API Wrapper` modules. This procedure was explained in section 5.6.

### 6.4 Registering the CreateTestAction

Though the action is fully implemented, it will not work until it is registered in the `System FileSystem`:

```xml
<folder name="Actions">
  <folder name="JUnitTests">
    <file name="cz-muni-fi-dbunit=CreateTestAction.instance"/>
  </folder>
</folder>
```
The Actions folder is a global repository for actions in the system, and it contains subfolders that categorize actions into ad-hoc categories. An instance file stands in for an instance of an object. An instance file typically says what class it is an instance of via its class name, as shown in the code snippet above. An instance file may create its instance from any Java class with a default constructor, or by calling a static method on a class [11].

Although the fragment of code above registers the action in the System FileSystem, it is not displayed anywhere in the GUI, and so the user is not able to click it. The next step is to display the action in the main menu and in the files’ popup menu. The following snippet displays the CreateTestAction in the Tools menu:

```
1 <folder name="Menu">
2   <folder name="Tools">
3     <file name="cz-muni-fi-dbunit-CreateTestAction.shadow">
4       <attr name="originalFile" stringvalue="Actions/JUnitTests/cz-muni-fi-dbunit-CreateTestAction.instance"/>
5       <attr name="position" intvalue="150"/>
6     </file>
7   </folder>
8 </folder>
```

Ideally, the action should be in the Tools menu close to JUnit’s action. This may be achieved by using NetBeans’ ToolAction. The ToolAction is a metaaction that contains other actions. The problem is that the DbUnit module would have to declare the action in its Manifest:

```
1 Name: cz/muni/fo/dbunit/CreateTestAction.class
2 OpenIDE-Module-Class: Action
```

The empty line can not be omitted because the Manifest’s sections need to be separated this way. In the DbUnit module, it was not implemented this way because it adds a lot of overhead to the system and is considered deprecated.1

Now, the action is registered and displayed in the menu bar. The last thing to do is register it in the popup menu in the Project window. This can be done as follows:

```
1 <folder name="Loaders">
2   <folder name="text">
3     <folder name="x-java">
4       <folder name="Actions">
5         <file name="cz-muni-fi-dbunit-CreateTestAction.shadow">
6           <attr name="originalFile" stringvalue="Actions/JUnitTests/cz-muni-fi-dbunit-CreateTestAction.instance"/>
7           <attr name="position" intvalue="2950"/>
8         </file>
9       </folder>
10     </folder>
11   </folder>
12 </folder>
```

This code adds the `CreateTestAction` to Java files’ popup menu. There is a shadow file registered. The shadow files are similar to Unix symbolic links in the operating system’s filesystem—it is a link to an instance file. Shadow files are used only when a single instance of an object is needed, but it must be registered in multiple folders. Shadow files can also point to real files on a disk. For example, the `Favorites` tab in the NetBeans IDE uses shadow files to link to real directories on the disk.
Chapter 7

DataSet Generator

7.1 Creating DataSets

The data set generator is a simple class through which the user is able to export data sets to XML files. As mentioned in section 2.4, there are more types of data sets. Since all tests use the FlatXMLDataSet, the data set generator contains code for generating this type of data set. There are also a few commented lines that can be used to generate an XMLDataSet that is more verbose. Examples of these two types can be found in Appendix A. If the developer wanted to use an XMLDataSet, he or she would have to change the method getDataSet in every test. There are two options left for the user: exporting the whole database or generating only the chosen tables. In addition, the class QueryDataSet that allows the developer to make an SQL select that can then be added to the data set was used. It is possible to add as many tables as needed, or to select from tables as needed.

7.2 The Wizard

Since the generator is very simple, its wizard contains only one panel. This panel is created using the standard package chooser from the module Java Project Support. The bottom part of the panel contains the only component, namely a combo box for choosing a registered database connection. The list of connections is gained in the same way as in section 5.3.

The method instantiate is implemented in the same way as its counterpart from the class EmptyTestWizardIterator. There was no need to change anything. The last step is to register the newly created template in the System FileSystem. Since this data set generator is implemented in the same way as the EmptyTestWizard, the registration code is also very similar.
Chapter 8
Conclusion

The first objective of this thesis was to get to know DbUnit, JUnit, and unit testing in database applications. The second objective was to get familiar with the RCP framework NetBeans Platform and implement a module that would integrate the DbUnit with NetBeans IDE. The author studied much literature on both NetBeans Platform and DbUnit.

As for DbUnit, the only documentation available was its home page, its wiki page and a few general articles on the Internet. Unfortunately, the wiki has not been updated for several months. Regarding NetBeans, there was only one book in English available. On the other hand, very good reference materials and tutorials can be found on its web pages. Another useful source was Sun employees’ blogs. The author also attended the NetBeans Platform Certified Training, which was very useful.

For a developer, implementation of a module in the NetBeans IDE is very efficient. There are many wizards whose output can usually be used without changes. Using these, the programmer is freed from writing skeletons of classes, modifying layer files, and creating many XML descriptors. The use of Matisse GUI Builder appeared to be a highly-efficient way to create the GUI. The module was created with an emphasis on maximal reusability. This seems to be a good step because there were only four panels implemented, and these were reused as much as possible. Classes from the JUnit module were also reused so it was not necessary to implement everything from scratch.

The main objective of this thesis was to implement a module that would add the support for DbUnit to the NetBeans IDE. The module can create empty tests, tests for existing classes, and data set generators. The author would like to continue developing the module. It would be desirable to implement advanced features of DbUnit even though they are not used very often. The GPL version 2 was chosen for source codes and they are available on the project’s web page (http://code.google.com/p/nb-dbunit/).
Bibliography


Appendix A

Data Sets

- FlatXMLDataSet

```xml
<?xml version='1.0' encoding='UTF-8'?><dataset>
<EMPLOYEE ID="1" FIRSTNAME="John" LASTNAME="Doe" EMAIL="John.Doe@johndoe.net"/>
<EMPLOYEE ID="2" FIRSTNAME="Jane" LASTNAME="Doe" EMAIL="Jane.Doe@janedoe.net"/>
</dataset>
```

- XMLDataSet

```xml
<?xml version='1.0' encoding='UTF-8'?><dataset>
<table name="EMPLOYEE">
<column>ID</column>
<column>FIRSTNAME</column>
<column>LASTNAME</column>
<column>EMAIL</column>
<row>
<value>1</value>
<value>John</value>
<value>Doe</value>
<value>John.Doe@johndoe.net</value>
</row>
<row>
<value>2</value>
<value>Jane</value>
<value>Doe</value>
<value>Jane.Doe@janedoe.net</value>
</row>
</table>
</dataset>
```
Appendix B

FreeMarker Template

```freemarker
<#-- FreeMarker template (see http://freemarker.org/) -->
<#assign licenseFirst = "/*" >
<#assign licensePrefix = " * " >
<#assign licenseLast = " */" >
<#include "../Licenses/license-$(project.license).txt">

<#if package?? && package != "" >
package ${package};
</#if>

<#if methodTearDown!false>
import org.junit.After;
</#if>

<#if classTearDown!false>
import org.junit.AfterClass;
</#if>

<#if methodSetUp!false>
import org.junit.Before;
</#if>

<#if classSetUp!false>
import org.junit.BeforeClass;
</#if>

import org.junit.Test;
import static org.junit.Assert.*;

/**
 * @author ${user}
 */
public class ${name} {
  public ${name}() {
  }

  <#if classSetUp!false>
  @BeforeClass
  public static void setUpClass() {
  }
  </#if>

  <#if classTearDown!false>
  @BeforeClass
  public static void tearDownClass() {
  }
  </#if>

}</#if>
```

@AfterClass
public static void tearDownClass() {
}
</if>
@if methodsetUp!false>
@Before
public void setUp() {
}
</if>
@if methodTearDown!false>
@After
public void tearDown() {
}
</if>
@if testMethodsPlaceholder!false>
// TODO add test methods here.
// The methods must be annotated with annotation
// @Test. For example:
//
// @Test
// public void hello() {}
Appendix C

Contents of Attached CD

The attached CD contains the following items:

- source codes of the module;
- a PDF version of the thesis; and
- a \LaTeX source file of the thesis