University of Helsinki
Department of Computer Science

TJC — A Framework for Testing Java Classes

Jukka Viljamaa

Helsinki, June 1997

University of Helsinki
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND

TJC — A Framework for Testing Java Classes

Testing object-oriented software can be done on various levels of abstraction. *White-box testing* of operations can be accomplished by following their internal execution paths. *Black-box testing* of classes involves checking public interfaces of individual classes or clusters of related classes. *System-level testing* addresses object collaborations in the whole application.

Black-box testing combined with *regression testing* is central in object-oriented systems, because they are usually developed iteratively. This implies the need to reuse — possibly partially modified — test material. This report introduces a small framework for testing Java classes (*TJC*) which addresses these requirements. It concentrates on testing public features of individual classes, but later it can be extended to handle also other kinds of testing[1].

The main purpose of implementing the framework has been to gain experience on framework development and design pattern usage. *TJC* is not intended to be a finished testing tool. This report is a product of the FRED (Framework Editor for Java) project ongoing at the universities of Tampere and Helsinki.

The implementation of TJC relies heavily on the *Reflection API* of the *JDK 1.1* (`java.lang.reflect` package) [Sun97]. It provides flexible means to query members of classes, create objects from them, and send messages to the created objects.

The structure of TJC

The basic concepts of the *TJC* are represented as an OMT class diagram in figure 1. Names of the abstract classes and interfaces (classes that are meant to be specialized[2]) are given in *italics*. The framework is divided into four packages `tjc.Base`, `tjc.UserInterface`, `tjc.Generator`, and `tjc.Report` (light gray areas in the figure). References to application classes as well as to their features and instances are represented in the diagram as attributes of type `Class`, `Member`, and `Object` (see, e.g., classes `TestObject`, `ObjectReference`, and `Execution`).

The class `TestBrowser` is the core class of the *TJC*. It manages classes to be tested and test cases that are currently loaded into the system. It also provides centralized access to `ViewFactory` and `GeneratorFactory` to create user interface elements and test material providers, as well as to `Oracle` and `TestReportStream` to attain test results and report them. `TestBrowser` keeps also track of all objects generated so far, all object generators that can be used, and all creation rules that can be applied

[1] To test hidden features of classes the system could be extended by defining a new `SecurityManager` class that would not raise `IllegalAccessExcpetion` when accessing private members. The system could also be extended further to allow generation of test material based on analyzing the source code of classes to be tested.

[2] Default implementations for all abstract interfaces are located in corresponding packages.

when constructing new instances (these three associations have been left out from the diagram for simplicity).

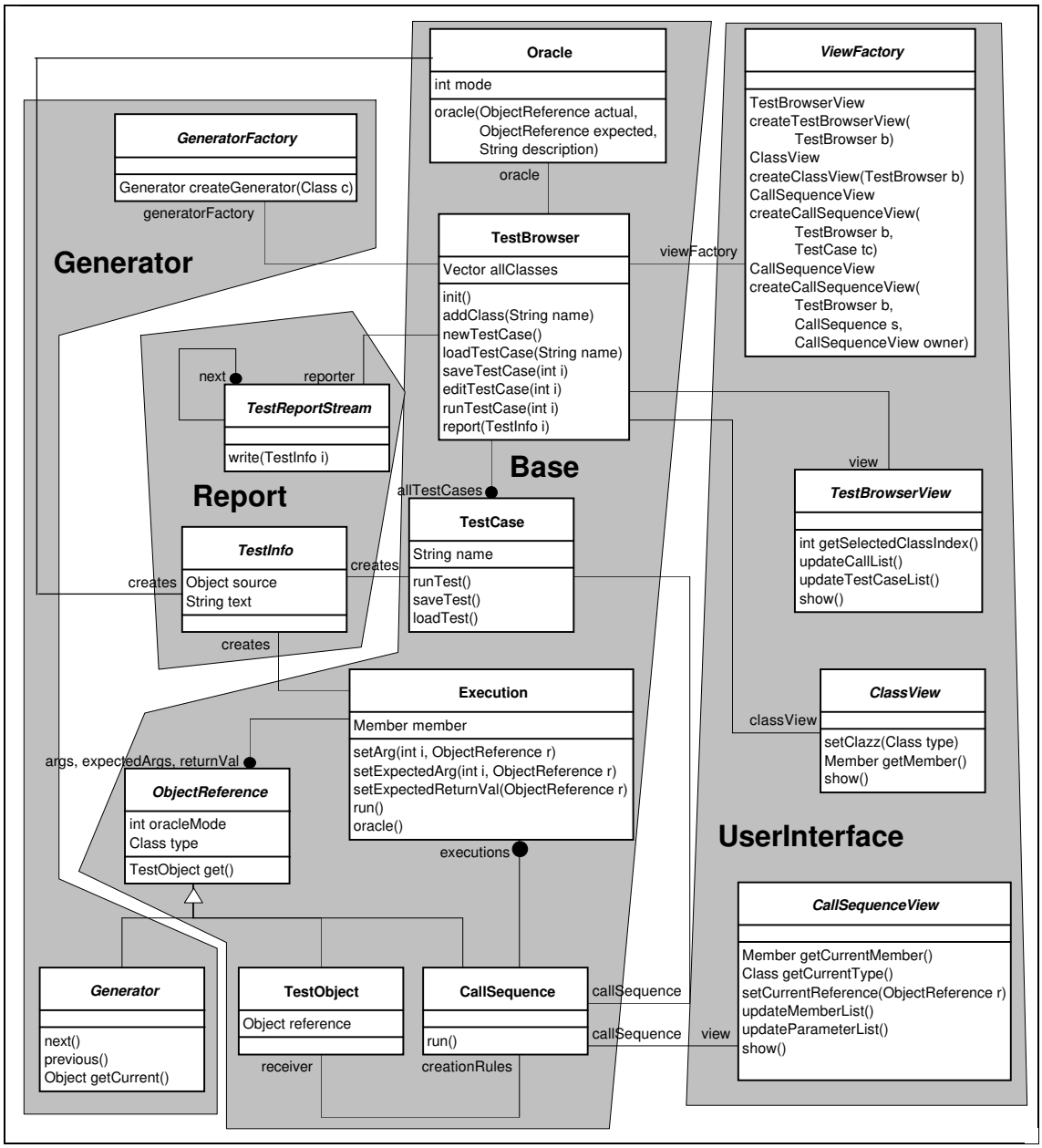

Figure 1: Basic concepts of TJC as an OMT class diagram

Tests are organized as test cases (`TestCase`) according to the target class of the tests. A test case consists of an ordered set of messages (`CallSequence`)[3]. For each message in a sequence there is an `Execution` object that identifies the actual and expected arguments together with the expected result object. They are all represented as instances of class `ObjectReference`. It stores object's `type` and an

[3] At the moment *TJC* handles only `Constructor` and `Method` members. In the future also `Fields` could be tested.

`oracleMode` field to indicate how to determine the success of the test. All messages in a particular call sequence have the same `receiver`.

`TestBrowser` object handles also the loading and saving of test cases. Test cases are stored using Java's serialization interface. Every object belonging to a test case implements the `Serializable` interface so that they can be written to a file stream. Those fields that are of type `Class` or `Member`, and are thus not serializable, are replaced with textual and numeric information (e.g., class name or member kind and index).

When the application evolves regression testing is needed. This can be accomplished by executing previously defined test cases. At the moment only exception information is printed out to the test report in case of a mismatch between test material and classes to be tested, so that user can adjust the test material accordingly. In the future some kind of automatic refinement of test cases could be provided. It should also be possible to refer to existing test descriptions as parts of new test cases, to make test cases nested, or to define loops and conditional branches in test cases. These features are not, however, yet implemented.

Test phases

Testing is partitioned into three phases in *TJC*. First in the *creation phase* constructor and methods to be called are specified together with their actual arguments and expected return and argument values after the execution. These can be specified to be references to existing objects using class `TestObject`, which is a wrapper class for actual references to application objects. Alternatively they can be generated by an instance of class `Generator` or created upon invocation by specifying `creationRules` as an instance of class `CallSequence`.

In the *execution phase*, messages are sent to the object under test in a predefined order (`CallSequence`) by calling the `run` method of each `Execution` in the execution sequence. In it the member to be used (message or constructor) is identified and the corresponding operation (`invoke(Object receiver, Object[] args)` for messages or `newInstance(Object[] args)` for constructors) is called. The arguments are fetched dynamically (`get`) in different ways depending on which kind of `ObjectReference` is in question. If it is an instance of class `Generator`, a new test object is created using operation `getCurrent`. If it is a reference to an already created object, that object is used directly (it can be, e.g., selected from a list of existing objects). If it is a call sequence defining the creation rules for the object, they are executed (`run`) and the resulting receiver object is used.

The final phase is called the *oracle phase*. In it the result object and the arguments are compared to their expected counterparts. Comparison is performed by the `Oracle` object based on different oracle modes (e.g., checks for identity or equality). `Oracle` provides information about the test results as `TestInfo`

instances. `TestCase` and `Execution` create additional `TestInfo` objects to indicate the progress of tests.

Test results are output through `TestBrowser`'s `reporter` object. It is a series of `TestReportStream` objects joined together one after another to form an arbitrary *Pipes and Filters* design pattern like combination [BMR96]. Various subclasses can be defined to implement the `TestReportStream` interface, for example to facilitate filtering according to different criteria, to branch streams, or to support different output variations such as text files, windows, or even hypertext documents. All stream objects handle different kinds of `TestInfo` objects that are generated by `Execution`, `Oracle`, and `TestCase` during the test case execution. The stream object combination to be used is defined initially by overriding `TestBrowser`'s `init` method or afterwards by setting the `reporter` reference.

Adapting TJC

TJC can be extended in many ways. The use of design patterns [GHJ95] is intended to make it flexible to adapt it to various application purposes. For example, the *AbstractFactory* design pattern is utilized to create all user interface elements. The outlook and features of user interface components can be changed by implementing appropriate interfaces (`TestBrowserView`, `ClassView`, `CallSequenceView`) and defining a new `ViewFactory` that creates instances of specific subclasses. Similarly user can define new subclasses of `Generator` to extend the system's test material generation capabilities.

User configures the system by overriding `TestBrowser`'s `init` method, so that correct abstract factory instances as well as `reporter` and `oracle` objects are created and `TestBrowser`'s corresponding fields are set to point to them.

Other design patterns applied in *TJC* include implementing `TestBrowser` as a *Singleton*, defining the fetching of actual object references as a *Strategy* pattern, and having the corresponding classes in the `tjc.Base` package to work as *Proxy* place holders for stored information before it is requested to be displayed. Using separate view objects can be also regarded as an instance of *Observer* design pattern with only one observer for each subject. Of course the system can be extended to allow multiple observers.

A sample application

A snapshot of a simple sample application (*Test Browser*) derived from *TJC* is depicted in figure 2. It is a similar test case management tool as the *TOBAC Test Case Browser* described in [SiN94] only with far more modest set of features. *Test Browser* has three main views *Test Browser*, *Class Members*, and *Test Case* representing classes and test cases, members of selected class, and properties of the test case that is being edited, respectively. The *Test Case* view lets user to specify constructors and methods to call by

selecting them from the lists of the *Class Members* view. Object references related to the selected call are also displayed so that user can define the way in which the objects are obtained when the method is invoked. User can also specify oracle modes for arguments and return values.

Figure 2: A sample application derived from TJC

In the figure, user is giving arguments for the method `wait` of class `Foo`. The second argument is already chosen to be generated with an `IntegerGenerator`. Oracle mode is selected to be *Equality* indicating comparison by calling objects `equals` method.

References

BMR96 Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., *A System of Patterns — Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

GHJ95 Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

SiN94 Siepmann E., Newton A., TOBAC: A Test Case Browser for Testing Object-Oriented Software. Proc. *1994 International Symposium on Software Testing and Analysis (ISSTA), ACM Software Engineering Notes*, Special Issue, 1994, 154-168.

Sun97 Sun Microsystems, Inc., JDK 1.1.2 Documentation, Internet: `http://java.sun.com/products/jdk/1.1/docs/index.html`, 1997.