TeeJay - A Tool for the Interactive Definition and Execution of Function-oriented Tests on Java Objects

Ralph Weires, Rainer Oechsle
University of Applied Sciences, Trier, Germany

weiresr@fh-trier.de, oechsle@informatik.fh-trier.de

Abstract

This paper describes the testing tool *TeeJay* whose main purpose is the function-oriented testing of Java objects with a particular support for remote RMI objects. TeeJay offers the possibility of defining and executing tests in an interactive manner. A test case consists of method calls and checks. TeeJay is a capture/replay tool comparable to many test tools used for testing graphical user interfaces. A test is defined by recording all interactively executed method calls and checks. The method calls can be selected for execution in a way similar to the way method calls are executed in BlueJ.

Whereas BlueJ is an educational development environment with a focus on the design aspect, TeeJay is a corresponding tool for testing.

1 Introduction

An important area of computer science education is software development. Teachers should emphasize right from the beginning that programming is not equal to software development, but only a part of it. Other important activities of the software development process are analysis and design on one hand and program testing on the other.

BlueJ (Barnes and Kölling, 2004) is an educational development environment with a focus on the design aspect. TeeJay is a corresponding tool for testing which can be used to interactively create and execute tests for Java programs with the help of a graphical user interface and without the need of writing any test code.

The available building blocks that can be used in a test are classes, existing objects, existing primitive variables, and existing remote resources. All these building blocks are visualized by trees. These trees contain the methods that can be applied to each existing class and object. Tests in TeeJay can simply be composed by interactively selecting methods or constructors that should be invoked within a test. It is also possible to specify conditions for the attributes of the objects or for the return value of a called method. To realize this functionality, TeeJay makes extensive use of the *Java Reflection API*.

TeeJay requires a Java Runtime Environment (JRE) of version 1.4 or higher. It can be freely used and copied under the conditions of the GNU General Public License (GPL). The whole underlying diploma thesis (Weires, 2003) as well as the software itself are available on one of the author's web pages (at `http://www.ralph-weires.de/stud-da.php`).

2 Usage Overview

TeeJay offers two working perspectives to the user, one for the definition of tests and the other one for the execution of existing tests. In this paper we will only show some sample parts of the user interface in order to give a brief overview over the main usage and functionality of the program.

2.1 Definition View

The definition view shows the current test environment which consists of classes, objects, primitive variables, and remote resources. Remote resources are represented e.g. by RMI registries and are used to obtain references to remote RMI objects. Once a reference to a remote object has been obtained, remote objects can be used like any other local object.

However, operations on RMI objects are automatically called with a timeout. An RMI call is cancelled after a certain (user-defined) time has elapsed without the method call having returned. Thus, we take into account possible disturbances of the network connection to the remote service.

The views for classes and objects show all members (attributes and methods) which can be used in a test. The classes view will now serve as an example to explain how the methods can be used. Figure 1 shows the view of some sample classes in the environment. The classes are displayed in a tree structure, sorted by their packages. The child nodes of a class

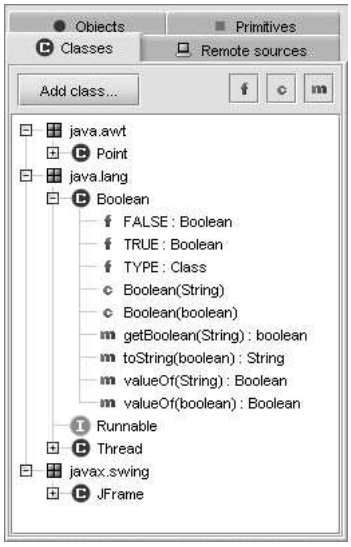

Figure 1: Classes view of the test environment

node represent constructors, class attributes (static attributes in Java), and class methods (static methods in Java). Figure 1 depicts these members for the class `Boolean`, because the node representing this class is currently expanded. TeeJay can, however, only access members with the modifier `public` - so bypassing the defined visibility for the members of a class is not possible in the current version of TeeJay (although this might change in a future version). The available members are visualized by their respective declaration. They can be used in tests. In the example it would thus be possible to call e.g. the static method `getBoolean(String):boolean`.

To actually call a method, the required parameters have to be specified. Any objects and primitive variables, which are currently present in the test environment and which were gained by the execution of previous operations, can be used. They can be referred to by their identifier. Besides such arguments coming from the test environment it is also possible to define fixed values for a call. In the case of a primitive parameter every literal of the respective data type can be used. In the case of an object parameter, `null` can be used or a literal string, if the parameter type is compatible with the class String. If the return type of a called method is not `void`, it is additionally possible to assign the return value of the called method to a variable of the test environment. The identifier of the variable has then to be indicated.

Instead of assigning the return value of a method call to a variable, it is also possible to check certain conditions for a return value. This option is mainly given for primitive values. Such return values can be compared with a type compatible reference value (which can e.g. be again a value out of the test environment), using one of the usual comparison operators like `==` oder `<=`.

After the complete specification of a method call, the method is actually called using the Java Reflection API. It is thus possible to compose the desired test environment piece by piece and to optionally check certain conditions. The result of every method call is displayed in

a logging window. In case of a thrown exception during the execution of a method call, the respective exception object is displayed in the logging window together with its stack trace to support debugging.

If a class is needed in a test, the class has to be added to the test environment. This is done by indicating the fully qualified class name. The classes view can so be filled as shown in Figure 1 for several sample classes and interfaces. The user is thus enabled to test class methods, or to construct her test environment by creating objects which will later be used in tests.

In order to allow TeeJay to access the required classes, the user has to specify the directories and libraries (jar or zip archives) that will be searched. Only compiled class files are needed, there is no source code required. For RMI objects it is also possible to enable dynamic class loading, thus making the local presence of the class files optional. To make use of this option, it is however needed to start the program with some special options which are not further described here.

The objects view which is shown in Figure 2 can be used in a way quite similar as the classes view. For every object, all public instance attributes and methods (non-static attributes and

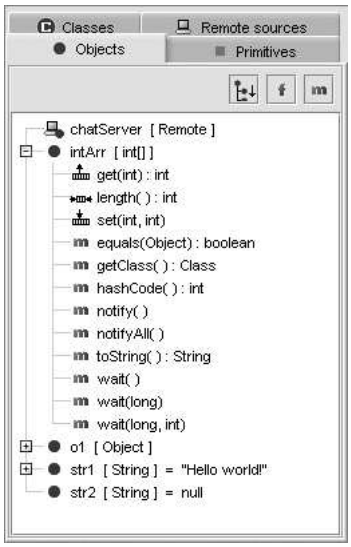

Figure 2: Objects view with pseudo array methods

methods in Java) are displayed. To get a better overview it is possible to hide the attributes, the methods, or all inherited members.

The type of a reference to an object gained by a method call depends on the formal return type of that method. The fields and methods of an object that are displayed, and are thus available, depend on the current reference type. The type of a reference can be interactively changed by an explicit cast operation.

If an object reference represents an array, TeeJay offers some pseudo methods which do actually not exist. These methods represent the reading and writing of an element of the array as well as reading the array length. The types of the parameters and return values of these methods depend on the array type. As an example, Figure 2 shows the methods get, set, and length for an array of int.

For testing purposes, TeeJay provides a kind of capture/replay function which can be used for the definition and execution of tests. To define a test, the program can be switched into a recording mode, in which all operations interactively executed by the user within the test environment are recorded. After having finished the recording, the recorded sequence of operations is available as a test case which can later be replayed.

It is additionally possible to define test suites which are composed of recorded test cases as described above or other test suites. Complex test trees consisting of many test cases can

thus be created. A test suite can be executed later by a single mouse click. For every test case within a test suite, it is possible to specify the number of times the test has to be repeated.

All defined tests can be saved persistently. This is why they can be reused in future sessions. To run one of the available tests, TeeJay provides a separate view (besides the definition view) which will be described in the next subsection.

2.2 Test Run View

The second perspective of TeeJay's user interface is used to select one of the available tests and run it. Every defined test can thus be executed again after the source code of the tested classes has been modified. Therefore TeeJay is suited for regression testing. A regression test checks whether all the functions that have been successfully tested in the past, still work after the code has been modified.

The execution speed of a test can be adjusted by the user. The progress of a started test can be observed during its execution. The results are displayed in two ways:

1. In the detailed view the result of every single step is displayed with all the details like thrown exceptions and stack traces in case of an error.

2. The summary of the results of the test run so far displays some statistics to give a rough overview over the current progress. Figure 3 shows an example of such a summary after completion of a test run. These statistics contain information about the number of executed tests, operations and checks (checks are operations, too). For the checks it is also displayed how many of them were executed successfully and how many failed. After a test run has been terminated, there is a single summarizing message being displayed to inform the user about the result of the test run. The message shown in figure 3 appears only if the test run was completed successfully (it was not aborted neither by the user nor by a failed operation), and if additionally all checks have passed.

3 Related Work

We know a few tools which have a functionality that is comparable to TeeJay. This section takes a look at the differences to these tools and the consequential pros and cons in comparison with TeeJay.

3.1 JUnit

JUnit (Beck and Gamma, 1998) is certainly one of the most common tools for testing Java programs. Like TeeJay, JUnit is used for the creation of function-oriented tests. It has a simple structure and is very versatile. It requires the definition of tests by directly writing test code. This has the advantage that there are no restrictions for the definition of tests, whereas TeeJay does not allow the definition of control structures, e.g., in tests.

Figure 3: Summary of a test run

The fundamental difference (besides the graphical user interface) between these two tools is that in JUnit tests are defined *statically*, whereas in TeeJay they are defined *dynamically*. Dynamic test definition means that all test steps are executed immediatly after their definition.

Hence in TeeJay it is possible to observe the results and effects of every step already at definition time. The user is thus provided with a test environment which is always up to date, making it possible to analyse the progress and the effects already during the definition of a test. Errors can therefore be detected immediately. We believe that especially for programmer novices this feature can be very helpful.

Another difference between the tools is that the single test cases in JUnit are independent from each other and can therefore be executed in any order without any problems. Thus every test case represents a self-contained test.

Tests in TeeJay may on the contrary have dependencies. If an object is e.g. added to the test environment in one test, it is usually still available after that test has finished. In this way tests can be created which are based on each other, making it possible, e.g., to reuse a basic test in more than one test scenario.

However, this feature of TeeJay has the disadvantage, that, during the execution of a test suite, a failure of a single test case may cause the failure of the whole test suite. This is different in JUnit because of the independency of the test cases.

3.2 Exacum

We found the commercially available testing tool Exacum (`http://www.ist-dresden.de/products/Exacum/`) only when the development of TeeJay was nearly finished. Even though TeeJay was not influenced by Exacum, there are some similarities between these two tools. But there are also some important differences.

Exacum is not only comparable to TeeJay, but also to JUnit. In contrast to JUnit, the definition of tests in Exacum is also done using a graphical user interface. Like TeeJay, Exacum offers the possilitiy to the user to define simple test steps like constructor or method calls as well as checks as part of a test case. However, this definition is done statically in Exacum, whereas it is done dynamically in TeeJay, as already mentioned in the previous subsection. The test steps are actually executed only if the user starts a test run. This happens after a test has been completely defined. So there is no execution of any test step during the definition of a test.

In this regard Exacum is similar to JUnit. Hence, the pros and cons mentioned in the previous subsection do not have to be repeated here. Futhermore, like in JUnit, different test cases are independent from each other.

To sum up, Exacum is like JUnit but has a graphical user interface for the definition of tests like TeeJay. JUnit offers more flexibility because the test steps have to be programmed in Java. TeeJay supports especially the testing of remote RMI objects, whereas Exacum mainly focusses on servlets and EJBs (Enterprise Java Beans).

3.3 BlueJ

BlueJ (Barnes and Kölling, 2004) is originally not a testing tool, but a development environment for Java whose main purpose is the support of teaching object-oriented principles to programming beginners. TeeJay adapts the philosophy of BlueJ to the world of program testing. Like in BlueJ, TeeJay users work on classes and objects, e.g., by interactively selecting methods to be called. Also like BlueJ, TeeJay visualizes the available classes and objects as well as their methods and attributes.

However, during the development time of TeeJay, BlueJ has been enhanced with components that provide testing support. This has been achieved by integrating JUnit into BlueJ (Patterson et al., 2003). Hence, it is meanwhile possible to define unit tests in BlueJ in a similar manner as in TeeJay. Besides the possibility of explicitly writing JUnit test code, it is

possible to interactively create a corresponding test class for a class represented in the UML class diagram view of BlueJ. For such test classes, test methods can be defined by recording a sequence of interactively executed calls of constructors and methods in the environment. Also, similar to TeeJay, it is possible to define certain conditions for the return values of called methods while a test is being recorded. These conditions are tested when the created test methods are run afterwards. The graphical user interface for executing tests is similar to the one known from JUnit.

Although the testing features of BlueJ and TeeJay are based on the same idea and are thus similar in many ways, there are also some significant differences. BlueJ supports test fixtures. A test fixture is an environment that is established before a test run is started. A test fixture can be reused for many test cases. TeeJay does not support test fixtures, although they can be emulated by recording additional preparation and cleanup tests which can then be used with different test cases inside of test suites.

Another advantage of the testing features of BlueJ is due to the integration of JUnit into BlueJ. The tests created by BlueJ are Java classes suited for JUnit. It is thus possible to run them also in the traditional JUnit environment - the BlueJ environment in which the tests were created is not needed for the execution. On the contrary, TeeJay is always needed to run the tests created with it. Furthermore, the created tests in BlueJ can be manually edited at a later time.

In TeeJay, there are no source files needed for the classes being tested, whereas in BlueJ, it is only possible to define test classes for those classes whose source code is available (unless the test code is written manually). In general, methods or constructors of classes cannot be interactively called in BlueJ unless the source files are available. On the contrary, in TeeJay classes can be displayed and used without the need of knowing the corresponding source code.

Another difference between the two interactive testing environments is the graphical representation of classes and interfaces. In BlueJ, UML class diagrams are used, whereas classes, interfaces, and objects are represented by trees in TeeJay (see subsection 2.1).

Furthermore, TeeJay was developed with a special focus on testing remote RMI objects. As far as we know, this is not possible in BlueJ. In addition, after having found a remote object in an RMI registry, it is possible in TeeJay to load the class file for this object dynamically over the network.

Finally, some features of TeeJay regarding the definition of tests are not possible in BlueJ (unless the test code is written manually). E.g., in BlueJ it is not possible to save the return value of a called method in the test environment if this value has a primitive type. This is mainly because the BlueJ environment only manages objects, not primitives. Therefore, it is not possible to create a test which checks whether the (primitive) return value of two methods, called one after the other, is the same.

3.4 JavaCHIME

Like BlueJ, JavaCHIME (Tadepalli and Cunningham, 2004) is more oriented towards teaching than testing purposes. It is another tool which allows direct interaction with objects and classes like BlueJ and TeeJay do. It is, however, currently under development and can not be tested by the authors yet. Thus, we are not able to provide further information.

4 Possible Usage of TeeJay in Practical Education

TeeJay can be used in practical education as a valuable tool for programmer novices. It provides a simple visual overview of the classes and objects currently in use, and enables the students to directly see the effect of a performed action such as a method call. TeeJay could thus help to understand important principles of object oriented programming in a similar way as BlueJ does.

More important is TeeJay as a tool that fosters the integration of testing into the software development process right from the beginning of computer science education. We believe that this is as important as the integration of software modelling techniques from early on. The students should be encouraged to create tests already for their small exercise programs that they have to write in their first year. After changing the underlying code, the same tests can be used again to perform a regression test without any more work to do. Students will discover that some of the test cases will not work any more. If they did not expect that the code changes have such an impact to their test cases, they will gain some important experiences and insight into the area of software development. Especially, they will get used to the testing process and they will achieve a better understanding of the eminent and increasing importance of testing software thoroughly before actually using it in a running system.

The achieved effect of using TeeJay in practical education could be evaluated i.e. by separating the students into two groups: one that is teached to work and test with TeeJay and another one that does not use TeeJay. We assume that the programs created by the first group will be better (less bugs, more robustness), since these programs will be better tested than those of the other group.

In order to get significant results, the experiment has to be carried out with larger student groups working on a number of different exercises. To realize the comparison of the student programs, a tool for the automatic assessment of the student programs (such as those described e.g. by Reek (1989) or Jackson and Usher (1997)) would be very helpful.

5 Summary and Outlook

TeeJay is a testing tool for programmer novices that offers the possibility of defining and executing tests in an interactive manner. A test case consists of method calls and checks. TeeJay is a capture/replay tool comparable to many test tools used for testing graphical user interfaces. A test is defined by recording all executed method calls and checks. The method calls can be selected for execution in a way similar to the way method calls are executed in BlueJ. TeeJay can be used as a tool for programmer novices that fosters the integration of testing into the software development process right from the beginning of computer science education.

Up to now, TeeJay supports only the definition of tests by explicit specification of each step. These steps are basically at the level of source code statements. This way of test definition is well suited for simple tests, but is too time-consuming for larger tests. JUnit is much more efficient and flexible in this respect.

The usability of TeeJay can be improved significantly if test cases are derived automatically from a given higher-level specification. State charts are an example of such higher-level specifications. By applying appropriate traversing algorithms for state charts, TeeJay could automatically find the needed test cases to cover all possible transitions of a state chart.

Such an approach is described in the diploma thesis of Sokenou (1999) which deals with the definition of state charts for class testing. The work focusses on a technique for the inheritance of state charts in class hierarchies.

Apart from such fundamental extensions, some smaller improvements are also planned such as modifications of the user interface. An extended view of the different contents would be useful to give an even better overview of the test environment to the user. An example for this would be a deeper display of the attribute values of objects as can be seen in many advanced debuggers. Some currently not used abilities of the Reflection API could be used to access members with limited access modifiers.

A possibility to edit defined (recorded) tests would be another useful feature which is not available at the moment. A little fault made at definition time would no longer force the whole recording process to be repeated.

The addition of further remote resource types is another direction for future work. Cur-

rently, only RMI registries are available as remote resources. Other possible remote resource types that can be made available for TeeJay are *JINI Lookup Services*, *Java Spaces*, *CORBA Naming Services*, and *Web Services*.

References

David J. Barnes and Michael Kölling. *Objects First with Java: A Practical Introduction using BlueJ*. Prentice Hall / Pearson Education, 2nd edition, 2004. ISBN 0-13-124933-9.

Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3 (7):37–50, July 1998.

David Jackson and Michelle Usher. Grading Student Programs using ASSYST. In *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pages 335–339, 1997.

Andrew Patterson, Michael Kölling, and John Rosenberg. Introducing Unit Testing with BlueJ. In *Proceedings of the 8th conference on Information Technology in Computer Science Education (ITiCSE)*, 2003.

Kenneth A. Reek. The TRY System - or - How to Avoid Testing Student Programs. In *Proceedings of the 20th SIGCSE Technical Symposium on Computer Science Education*, pages 112–116, 1989.

Dehla Sokenou. Ein Werkzeug zur Unterstützung zustandsbasierter Testverfahren für JAVA-Klassen. *Softwaretechnik-Trends (in German)*, 19(1):37–50, Februar 1999.

Pallavi Tadepalli and H. Conrad Cunningham. JavaCHIME: Java Class Hierarchy Inspector and Method Executer. In *Proceedings of the ACM-Southeast Conference*, pages 152–157, April 2004.

Ralph Weires. Entwurf und Implementierung eines Werkzeugs zur interaktiven Definition und Ausführung funktionsorientierter Tests für lokale und über RMI nutzbare Java-Objekte. Diplomarbeit (in German), Fachhochschule Trier, 2003.