AUTOMATIC TESTING TOOL FOR JAVA ADTS

Miguel Ángel Quintans Rojo
Professor at the Dpt. de Ciencias de la Computación
Universidad de Alcalá

Virginia Escuder Cabañas
Professor at the Dpt. de Automática
Universidad de Alcalá

ABSTRACT
The inherent properties of Abstract Data types are a good basis for systematic testing and for decoupling design and development of software. We propose checking the implementation of Java Classes for Abstract Data Types using the Java Class Interface Definition and an Oracle of Test Cases that specify expected results. This task is performed by a software tool proving for automatic Unit Testing.

KEYWORDS
object-oriented software testing, unit testing, abstract data types

1. INTRODUCTION
It has been known for several decades that Testing software is a critical and very costly activity in the software development process. It is a tedious task that people feel reluctant to perform, specially under time pressure. It is, therefore, important to automate the process of testing to encourage its practice. Particularly, regression testing can be achieved using automatic tools that will keep track of all tests cases and will launch all of them whenever a new test should take place following software modifications.

We support the usage of Abstract Data Types (ADTs) as building modules for constructing software systems. This is the basis for supporting the principle of “information hiding” [8], effective software maintenance and the participation of different teams for the design and development activities. For similar reasons, in our approach, testing does not exist by itself and it is not based on code internals because we believe it limits tests reutilization as code changes thus making testing an even more stressing task. Testing should be therefore, tightly related to the design methodology: ADTs, and whenever possible, test should check an implementation against its specifications.

There are some important efforts to process formal axiomatic specifications of ADTs and check its implementation during execution, i.e. using term rewriting techniques [3]. The benefit of these kind of approaches is that the specification itself acts as an “Oracle” for the tests, without needing test cases design. But it also requires a very knowledgeable team of software engineers to make it, and a quite exceptional environment of use.

Our approach requires human design of test cases and it may be more tolerant and flexible in the axiomatics required for the specification of the ADTs, so we believe it is potentially usable for people who should have a level of expertise we could expect to find in industrial or commercial sectors.

The ideas around this testing tool and some previous implementations come from the authors experience in the 80's [11] and 90's while developing software in industrial environments. The current ongoing implementation targets for Java classes, as this language is becoming very popular and it has some special features that can be advantageous for the methodology the tool is based upon.
2. DESCRIPTION OF THE TOOL AND ITS ENVIRONMENT

Figure 1 shows a block scheme of the tool. We use a simple syntax of imperative statements containing invocations for the different Class Methods together with its expected results (Oracle). These are test cases specifications. It then links to the actual code of the Class implementation and executes these test cases, showing any discrepancies upon execution.

2.1 Interface Class

This is an abstract class in Java that will serve as the external specification of the ADT. It is what the designer does and it describes every method of the type by specifying its name, arguments and function type (domain and range). He/she may describe the behavior of the ADT using either algebraic specifications or natural language in Javadoc comments, that can later be used to generate documentation in HTML format. This interface class contains all necessary information to code it and should be used by the programmer in charge of its development. It will compile together with the code. This ensures that there will be no mistaken names or missing methods as it will compile with the actual code ensuring full match.

2.2 Test cases Specification

This is a text file containing any invocations needed to create instances of the class and perform transformations on it together with calls to informational methods that should check the effect caused by the former transformations. These checking sentences are boolean test sentences that are expected to be TRUE, like

\[
\begin{align*}
  t &> ! s.isEmpty() \\
  t &> s.top() == new Integer(7)
\end{align*}
\]
where “t >” indicates a test operation. Upon execution, the test monitor will check for a “true” result. Figure 2 shows an example.

Test cases are written, typically by the designer at the same time he/she defines the ADT writing the Interface Class. Obviously, these test cases are black box style.

```
Test: Pila;

Pila s = new Pila();
s.push (new Integer (5));
s.push (new Integer (8));
s.push (new Integer (4));
t> ! s.isEmpty();
t> s.top() == new Integer (4);
s.pop();
t> s.top() == new Integer(8);
Integer siete = new Integer (7);
s.push (siete);
t> s.top() == siete;
s.pop();
t> s.top() == new Integer(7);
s.pop();
t> ! s.isEmpty();
t> s.top() == new Integer(5);
s.pop();
t> s.isEmpty();
```

Figure 2. Test Cases

### 2.3 The Module under test

This is the actual Java class implementing the Interface specifications written by the programmer. It is the Unit under test. It should not use global variables, not have side effects, etc. That is, it should follow recommendations for good coding practices.

In case the programmer uses some special feature in the code that is considered worth testing, then a specific test case for it can be inserted into the test cases specifications to ensure better test performance. Consequently, testing can flexibly be a mix of black box and other test strategies if necessary.

### 2.4 Test results

Figure 3 shows a sample of a test execution. The test monitor executed the specified test cases against the module under test. The output shows all test cases evaluated clearly identifying those whose result didn't match the expected value.

```
Pila s = new Pila ();
s.push (new Integer (5));
s.push (new Integer (8));
s.push (new Integer (4));
! s.isEmpty () ;
s.top () == new Integer(4);
s.pop () ;
s.top () == new Integer(8);
Integer siete = new Integer (7);
s.push (siete);
s.top () == siete;
s.pop () ;
t> ! s.isEmpty () ;
t> s.top () == new Integer (5);
s.pop () ;
t> s.isEmpty () ;

>>> Error: The result is 8
s.pop () ;
! s.isEmpty () ;
s.top () == new Integer (5);
s.pop () ;
s.isEmpty () ;
```

Figure 3. Test results
3. TEST MONITOR GENERATION

The generation of a Java Class that will execute the specified testing, called “test Monitor”, is performed by the Test Monitor Generator. This Monitor Class is compiled and executed by the user (or by some integration testing tool automatically) importing the Class under test observing test results (Figure 3). The same monitor may be used for several test sessions. Generation is only required if the Unit under test is a completely new one or if its description has been changed and consequently, the test cases are changed. This tool has been built using yacc and lex under a Linux system and the C programming Language. Makefiles are used as integration testing tool as it is sufficient to launch a hierarchy of unit tests to incrementally test an ADT hierarchy that implements a software system.

4. CONCLUSION

Using ADTs for building Software Systems is a very powerful philosophy to support Software Engineering but it requires some expertise. The tool presented here is a simple one for Unit Testing. Its success is depending upon the methodology itself, however it can be seen as an added stimulus to apply it. It will also help on the automatic execution of regression testing during software development cycles for improving maintenance and productivity. Planned next versions will support exception handling and threads control.

REFERENCES