# Assessing with a unit test framework: testware construction strategies

**Mike Lance**
School of Computing
Christchurch Polytechnic Institute
of Technology
lancem@cpit.ac.nz

**Ranran Bian**
School of Computing
Christchurch Polytechnic Institute
of Technology
rab446@student.cpit.ac.nz

**Amitrajit Sarkar**
School of Computing
Christchurch Polytechnic Institute
of Technology
sarkara@cpit.ac.nz

This quality assured paper appeared at the 1st annual conference of Computing and Information Technology Research and Education New Zealand (CITRENZ2010) incorporating the *23rd Annual Conference of the National Advisory Committee on Computing Qualifications*, Dunedin, New Zealand, July 6-9. Samuel Mann and Michael Verhaart (Eds).

## Purpose

We extended a unit testing framework so that it could automatically mark programming assignments. Assignments change, programming languages change, and students' work varies greatly. We sought a software architecture which would cope with these changes.

## Approach

Our first design goal was to make it easy to author the tests. A domain-specific language (Ghosh 2010) was created to allow a test author to think about just describing the data structures and methods the student was required to construct. This 'minilanguage' (Raymond 2004) allowed definition of a higher-level language to specify the appropriate methods, rules, and algorithms for the task at hand. It reduces global complexity relative to a design that uses hardwired lower-level code for the same ends. Specifically it hides the details of assertions within the unit testing framework (Beck 2002) and hides the reflective code (Sobel & Friedman 1996) required to detect and run the student code from a program which observe and modify properties of its own behavior.

Shearing layers (Foote and Yoder 2000) were used to further factor the system so that artifacts that change

at similar rates are together. A base class contains the methods which ran student code and accessed data within their work, a subclass contained the code which asserted that desired features should be present and subsequent subclasses marked specific questions.

Because the programme was embedded within the architecture of a unit testing framework we got 'for free' a generic engine to run the our tests of student code and a GUI to present the results of tests. The Jade programming language specific version of xUnit (JSC 2009) provided a base JadeTestCase class from which from we inherited the assertion methods that determine whether a unit test fails or succeeds. It allows a unit test run to be structured with methods to set up data in a known state (the test fixture) before a unit test starts and to tear down that data when the unit test has completed. The associated JadeTestListenerIF interface (the listener object) controlled the way the results are captured, displayed, and analyzed. And finally the JADE development environment enabled execution of a selected unit test method or all unit test methods for a selected class by pressing the F9 key.

## Findings

The overall result is a domain specific language which allows easy authoring of tests to mark programming assignments. Here is code which defines what the student should have done. This is isolated by shearing layers in successive super classes from the code which detects what the student actually did and the code which compares the teacher's desires and the students' work and assigns marks or provides detailed feedback of what is incorrect and how to fix it.

Automated marking of programming assignments reduces the workload of teachers and increases the speed at which students get feedback about their coding. The immediacy of the feedback starts their test addiction. We are beginning to explore variations in the type of feedback the software gives to students to see what effectively prompts them to them improve their code. Future work will involve porting the system to another programming language.

## References and Citations


Beck, K. (2002). *Test Driven Development: By Example*. Boston, MA: Addison-Wesley.

Foote, B. & Yoder, J. (2000). Big Ball of Mud. In N. Harrison, B. Foote, and H. Rohnert (Ed.), *Pattern Languages of Program Design* (pp 653-692).Reading, MA Addison-Wesley.

Ghosh, D (2010). *DSLs in Action*. New York, NY: Manning Publishing Co.

Jade Software Corporation Limited (2009). *Chapter 22 Using the JADE Testing Framework* in Developer's Reference Version 6.3.

Raymond, E. (2004). Minilanguages: Finding a Notation That Sings. In E. Raymond (Ed.)*The Art of UNIX Programming*. Boston, MA:Addison-Wesley.

Sobel J.S. & Friedman, D. P. (1996). An Introduction to Reflection-Oriented Programming. Indiana University. Retrieved May 11, 2010, from http://www.cs.indiana.edu/~jsobel/rop.html