Teaching with a Unit Testing Framework

Dr Mike Lance
School of Computing
Christchurch Polytechnic Institute of Technology
Christchurch, NZ
lancem@cpit.ac

In the NACCQ curriculum desk checking is used to introduce novice programmers to testing. It is argued that using a unit testing framework for subsequent teaching of testing is an equally effective teaching technique.

Keywords

Testing, desk checking, Unit Testing Framework, xUnit.

1. INTRODUCTION

Testing is a major time and cost factor in commercial software development - up to 50% of cost (Jones 2001). Because a wide range of practices and skills are required to produce well-rounded software testers a strong case can be made for widespread integration of testing topics throughout the whole undergraduate curriculum (Jones 2000). Many report however that testing is perceived very negatively by novice programmers (Canna 2001, Edwards 2003, Jones 2001).

In the NACCQ New Zealand Qualifications in Information and Communications Technology module prescriptions software testing is first addressed with an emphasis on desk checking. (PP490, PD500). Subsequent programming modules stipulate more general testing skills in learning objective that are variations of "design test data that will check that a program works to a specification" (PP590, PR50n, PR51n, PR60n, PR61n, PR62n, PR65n, PR72n). There is also a specialized module (QA600) with a comprehensive coverage of testing.

This paper will argue that the strategy of emphasizing a specific testing technique (desk checking) should be extended by also putting an emphasis in all programming modules on writing tests with a unit testing framework (xUnit). There is a strong need to get novice programmers competent and enthusiastic about testing and xUnit provides a good vehicle for doing this.

2. DESK CHECKING

Introducing novice programmers to testing via desk checking is a first experience which will leave a strong and lasting impression. Desk checking is very effective at teaching novice programmers to simulate and check the flow of logic in an algorithm. (The alternative to students proving the soundness of their own work is letting them handing code to a tutor and leave it up to the tutor to sort out if the code passes or not. This is not very empowered and the novice ends up being dependant on the tutor as tester.) It is a concern is that having learned desk checking in subsequent programming courses students will seldom use it unless they are required to. (In the authors 'other life' as a programmer he only resorts to desk checking when encountering a complex logical algorithm, written by somebody else, in a programming language he don't have access to. By preference he will use a debugger and set watch and break points.) Limited use of desk checking by advanced students may be an encouraging indication that the method effectively teaches mental simulation of the execution of a programme. There is a danger that students learn a negative message when the first formal software testing method they meet is not one they will actually use to write software. It is worthwhile spending the time to make clear to novice programmers that mentally desk checking is acceptable in many situations and is the eventual learning goal.

An emphasis on desk checking can also leave the impression testing comes after writing code.

Before a desk check can be carried out, an algorithm needs to be designed (in pseudocode, problem description language or a teaching language such as Pascal). After the algorithm is written down, a test situation must be defined and expected inputs and outputs must be determined. Only then can the algorithm be desk checked. If students repeatedly have the experience of designing a correct algorithm and then have going through a time consuming formal process which detects no mistakes they may learn that desk checking is an unnecessary nuisance and testing is an annoyance. Setting problems with subtle logical traps in the requirements or getting students to locate subtle bugs in code can help make the value of desk checking clear. The teacher of testing needs to also ensure that students continue to encounter situations where formally testing of non-trivial logic is required.

The wrong message about testing can also conveyed by the format in which students are required to carry out desk checks. Manual desk checking is an awkward exercise with rubbing out of incorrect results, repetitive writing of the same text, and much taping of pages together. The implicit lesson which this teaches is that testing is a dated and painful experience. (Programming students will seldom use a pencil, rubber and ruler except when doing manual desk checking.) Enforcing the use of awkward 1960s technology is also not necessary when it is so easy to create and record the results of a desk check in a spreadsheet. Using a spreadsheet for desk checking reduces many of the taxing mechanical components of the task and allows the student to concentrate on the actual testing.

3. THE JOY OF XUNIT

The biggest problem with desk checking as an introduction to testing is that it may teach students that testing is not fun. One important educational benefit of testing with a unit testing framework ('xUnit' from now on) is that the associated enthusiastic attitude towards testing. The claim is made that programmers will become 'test infected' and will 'love writing tests' and be 'amazed at how much more fun programming is'. (Beck & Gamma, 1998a) Part of this positive attitude towards testing is due to advantages that accrue when any sort of automated tests exist and help find bugs in code. With a firm foundation of tests, programmers become 'faster' and 'more productive'. (ibid) A particular reason why devotes of xUnit have a positive attitude towards testing is an altered emphasis on the place of testing within the software development life cycle. The recommended practice is to write the tests before coding to define requirements: "Add a test, get it to fail, write code to pass the test. Remove duplication" (Beck 2002 p5). Testing after coding will invariably identify failures and is a negative experience for the programmer. Coding after writing tests in contrast is a positive experience because the number of failing tests decreases and the increasing number of passing tests confirms that the code is working. This positive attitude associated with testing is of great value to educators seeking to teach testing.

'Test driven development' is also claimed to have a subtle impact on the quality of software design. Code has to be used by two 'clients': the system under development and the testing framework. This attention to developing a testable interface forces design for reuse. Code produced in this way is said to be 'reusable', 'less coupled' and 'highly decoupled'. (Beck & Gamma, 1998a) Once the break through of using code in two different situations has been achieved, subsequent reuse is said to be much easier. The final recommended step in the test driven development process is 'remove duplication'. The existence of unit tests and correctly working code is an 'essential precondition' for refactoring (Fowler, 1999, p89). The support and reassurance of working tests allows aggressive refactoring which can incrementally improve the design of code. The potential is that introducing xUnit appropriately into the software development life cycle can foster other positive changes to the quality of students' coding practices.

Another advantage of xUnit for the educator is that it is freely available for down load, is available in many different programming languages and has an associated large number of worked examples with extensive commentary. (eg Beck 2002, Pilgrim 2004). There are also discussion groups and Wikis where industry based xUnit users enthusiastically recount their experiences. This amounts to a rich and readily accessible set of teaching resources for teaching testing.

4. USING XUNIT

Using xUnit for testing involves setting up "microworlds" with test vocabularies specific to the application being tested. This is a 'test fixture' of known objects that can be accessed from every test method. A setup method gets called by the framework's main 'engine' before each test to create the objects to be tested. A teardown method is also called by the framework after each test method and the code put there should destroy all objects and generally clean up memory usage. This means that each test method can be assured a known set of object of known state. Test methods can then exercise functionality on these objects and check if methods are performing appropriately.

As a mature framework should (Foote & Yoder 1997) xUnit 'works out of the box' exposing its services with a very compact interface and generally has a 'gentle learning curve'. A lot of effort has been put into minimizing the amount of code that needs to be written to create a test. A student can start using xUnit to write tests after being given only a few lines of boiler-plate code. Pilot studies of students using xUnit in this manner have shown a marked reduction in code defects (Edwards 2003).

There are also a 'down side' to using xUnit. A framework by its very nature is complicated. Powerful and subtle object-oriented mechanisms are used as a matter of course when writing tests with xUnit. These techniques include extension by subclassing from inheritance-based class hierarchies, conforming to public interfaces, understanding and using run-time reflection mechanisms, the placement of code in different methods which will be called from a coordinating super class Template Method and generally programming according to the 'Open-Closed Principle' (Meyers 1988, p25, Martin, 1998) This level of understanding of object-orientation is not what was envisaged in the lower level NACCQ programming modules. To use xUnit requires either a major step of faith ('it just works') or a sophisticated understanding of object-orientation. Most tutorials about xUnit take a pragmatic approach and include lots of applied examples (eg Beck & Gamma 1998b). There are also more detailed articles explaining how xUnit works (Beck & Gamma 1999) but these are not promoted as introductory documentation.

The original vision for xUnit was of a simple and light weight framework (Beck 1994). Although the programming language roots of xUnit were in Smalltalk, the most popular implementation is JUnit which is written in Java. Alternative implementations in other programming languages also proliferate. Because these ports are created as open-source community projects the standard of implementation can vary widely. It is telling that Kent Beck, the 'father' of xUnit resorts to Python when discussing xUnit's underlying architecture (Beck 2002, Ch20). JUnit has accrued many additions over time that may be off putting to students. There are for example 37 variations of the assert statement to choose from. This complexity has ironically prompted some to refactor JUnit and produce simpler versions (Venners et al, 2003).

Even though xUnit may be off putting because of its apparent complexity, the alternatives are ultimately much more complex. Manual testing, by entering assorted values into a graphical interface, is time consuming, error prone, difficult for a tutor to verify and downright boring for the student to do. Automated testing tools which use a record and playback mechanism are expensive, difficult to use and often produce scripts which are 'too britle to be useful' (Kent 1994). Creation of rigorous automated testing software from scratch requires a reasonably high level of programming sophistication. Write good 'testware' from scratch is a difficult task for novice programmers still grappling with basic programming language syntax. The focus of the student's efforts can move rapidly from doing testing to writing the testware and the overall experience will be that testing is too difficult to be worth the effort.

5. CONCLUSION

It would be a "Good Thing" if novice programmers learned early on that testing is an easy process and not just an annoying 'add-on' to be done after writing code. Careful teaching of desk checking begins the educational process by teaching students how to simulate and verify the execution of an algorithm. Desk checking is a tried and proven technique. XUnit in contrast is a new and technologically innovative tool for testing. If novice programmers are taught test driven development with xUnit they may well discover the fun of testing and also become better programmers.

REFERENCES

Beck, K. (1994) Simple Smalltalk Testing: With Patterns. Smalltalk Report, October 1994,. Available on line at http://www.xprogramming.com/testfram.htm

Beck, K. & Gamma, E. (1998a) Test Infected: Programmers Love Writing Tests, Java Report, July 1998, Volume 3, Number 7) available on line at http://members.pingnet.ch/gamma/junit.htm

Beck, K. & Gamma, E. (1998b) JUnit Cookbook. Java Report, 1998. Available on line at http://junit.sourceforge.net/doc/cookbook/cookbook.htm

Beck, K. & Gamma, E. (1999) JUnit A Cook's Tour. Java Report, May 1999. Available on line at http://junit.sourceforge.net/doc/cookstour/cookstour.htm

Beck, K. (2002) Test Driven Development: By Example, Addison-Wesley, Boston, MA. Pre-publication copies available for download from http://groups.yahoo.com/group/testdrivendevelopment.

Canna, J. (2001) Testing, fun? Really? Using unit and functional tests in the development process. http://www-106.ibm.com/developerworks/library/j-test.html

Edwards, S.H.(2003) Teaching software testing: Automatic grading meets test-first coding. In Addendum to the 2003 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications. 2003. Available on line at http://people.cs.vt.edu/~edwards/pos27-Edwards.pdf

Foote, B. & Yoder, J. (1997) The Selfish Class in Chapter 25, Pattern Languages of Program Design 3 edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998

Jones, E L. (2001) Integrating testing into the curriculum - arsenic in small doses. Proceedings of the Thirty-second

Jones, E. L. (2000) The SPRAE Framework for Teaching Software Testing in the Undergraduate Curriculum. Proceedings of ADMI 2000(June 1-4, 2000).

Martin, R.C. (1996) The Open-Closed Principle. C++ Report Available for download from http://www.objectmentor.com/resources/articles/ocp.pdf

Meyer, B (1988) Object Oriented Software Construction, Prentice Hall.

Noonan, R. E. & Prosl, H. R. (2002) Unit Testing Frameworks. Thirty-third SIGCSE Technical Symposium on Computer Science Education, (February 2002), pp. 232-236.

Pilgrim. M. (2004) Dive Into Python. Accessed from http://diveintopython.org/ on 20 May 2004.

Testing Framework http://c2.com/cgi/wiki?TestingFramework

Venners, B. Gerrans, G. & Sommers, F. (2003) Why We Refactored Junit: The Story of an Open Source Endeavor. Available for download from http://www.artima.com/suiterunner/why.html