A TDD Approach to Introducing Students to Embedded Programming

James Miller
Electrical and Computer Engineering
University of Alberta
Edmonton, Alberta, Canada
+1 (780) 492-5580
jm@ece.ualberta.ca

Michael Smith
Electrical and Computer Engineering
University of Calgary
Calgary, Alberta, Canada
+1 (403) 220-6142
smithmr@ucalgary.ca

ABSTRACT

Learning embedded programming is a highly demanding exercise. The beginner is bombarded with complexity from the start: embedded production based around a myriad of C++ constructs with low-level elements integrated onto ever more complicated processor architectures. The picture is further compounded by tool support having unfamiliar roles and appearances from previous student experiences. This demanding situation often has the student bewildered; seeking for "a crutch" or the simplest way forward regardless of the overall consequences. To control this potentially chaotic picture, the instructor needs to introduce devices to combat this complexity. We argue that test driven development (TDD) should become the instructor's principal weapon in this fight. Reasons for this belief combined with our, and the students', experiences with this novel approach are discussed.

Categories and Subject Descriptors

K3.2 [Computers and Education]: Computer and Information Science Education

General Terms

Design, Verification.

Keywords

Embedded Programming, TDD, Education

1. INTRODUCTION

Introducing students to embedded programming is an especially demanding challenge. Complexity rains down upon the students from every conceivable angle. Modern embedded development uses object-oriented languages that utilize many high-level structures, while demanding highly efficient solutions. The student must wrestle with high-level abstraction mechanisms, such as function overloading, multiple inheritance, class and function templates, template parameters, etc. At the same time consideration must be made of many low-level concepts: specialized DSP-specific language extensions, direct hardware access, lightweight objects, explicit memory allocation control, combined with assembler code (inline and stand-alone) that is custom optimized for complex processor architectures.

The student is often left bewildered about how to merge often conflicting viewpoints of the programming universe. On the support or tool front, the students are required to gain familiarity with systems which take them far beyond their "knowledge-base" developed in a standard desktop environment into architecturally-aware compilers and cycle- and functionally-accurate compiled simulators. At the other end of the teaching spectrum, the instructor is also struggling to deliver the material. Embedded systems, if not used in a trivial manner, need to use custom hardware and industrial-strength software; each with steep learning curves. The tool support is complicated by embedded systems being real-time and performance constrained, making it difficult to freeze the entire execution context under the control of a debugger. The average student's concept of embedded debugging, via implanted *printf* or *cout* statements, becomes totally inappropriate as such statements require the transfer of control from the embedded target to a host-based tool.

This seemly wondrously complex environment of highly diverse, and often conflicting, components leaves many students befuddled. They are forced into seeking the lowest common denominator to survive. This survival instinct implies that instructors have difficulty in providing the students with opportunities to "see the forest for the trees" when it comes to undertaking programming tasks; and any intent to have them programming within the big picture, applicable across many processor families or producing software assets, gets quickly lost.

2. INTENT IS EVERYTHING!

What differentiates programming from mere coding or hacking is intent. Programming education needs to continually reinforce that programming is more than the production of some piece of code which approximates some functional objective on a particular processor. Programming is all about the production of systems with a wide array of characteristics. Unfortunately, all characteristics, apart from basic functionality, tend to disappear when students enter into "survival mode". Hence, the professor

ITiCSE 2007, June 25-27, 2007, Dundee, Scotland, U.K..

must introduce mandatory mechanisms, which force the student to consider the characteristics of what they are producing, and to keep on considering these characteristics until they become second nature to the student. Thus there arises the untenable educational equivalent of an industrial manager insisting a production team adopt a new process while experiencing severe problems during a major project. Further, while the production team can see that their future jobs depend on the continued usability of the product under development, the student is "learning on the job" and "knows" that on submitting the current assignment, the components will never be seen again.

Firmware engineers recognize the need for reliable embedded products, but are not eager to implement process improvements edicts-from-above. Greene [1] suggests that Agile's barely sufficient processes should be acceptable in such situations. We believe that this concept can be moved out of its industrial context with test driven development (TDD) playing a crucial role in forcing the student to consider the "big picture" characteristics. However, applying TDD in an educational embedded context is far from straightforward. In the desktop, object-oriented setting, we often concentrate on characteristics such as readability, correctness, robustness, extendibility and reusability. These remain important within an embedded environment; however this new domain tends to add additional, sometimes conflicting, characteristics such as efficiency and security.

The balance between these characteristics changes based upon the application requirements. For example, paraphrasing Dahlby [2], "Many embedded systems such as PDAs or cell-phones are high-volume, low-cost and low-margin requiring use of the cheapest components possible -- simple processors with small RAM and NVRAM/flash memories. This causes embedded systems software to trade off maintainability aspects such as portability, clarity, or modularity, for performance optimization aspects such as a small boot image footprint, a small RAM footprint, and small cycle requirements. The increased up-front software development costs and periodic maintenance costs are amortized by the high-volume sales, and outweighed by the continuous hardware cost savings of cheaper components." In other scenarios, embedded software and hardware components are built as (1) reusable assets or (2) as a product within an entire product line. The key to getting students to consider these characteristics is to describe them as a series of executable tests defined as system or acceptance tests and as function or unit-level tests. TDD is ideal from this perspective as the entire life-cycle is test-case driven.

Clearly, our work has many parallels with integrating "Test-first" practices into programming courses (e.g. [3, 4]). However, that other research tends to view TDD as a vehicle for introducing testing into programming courses, whereas we consider TDD as an approach that permits both the formalizing and extension of the limited testing already in place. In addition, we view TDD (in line with standard Agile theory [5, 6]) as a design process rather than a verification process. Thus TDD is utilized in two distinct modes within the classroom. (1) When the student writes the assertions (or constraints), the student is exploring the design space through incrementally building up a picture of the solution's requirements. (2) When the professor writes the assertions, it is done in the "expert customer role"; adding explicit constraints of the design of the system. Further, due to their executable example-based nature, these constraints provide an exact contractual statement, forcing students to adhere. However, because the constraints are added at a high-level, the student is still challenged to produce a solution which goes beyond basic functionality.

It is believed that embedded construction is a highly complex undertaking that requires several courses to adequately teach. Our TDD-based approach is designed to be a constant across all of these courses, and to extend back into pre-requisite courses such as desktop programming classes. However, in this paper we will concentrate upon our experiences over three years in teaching compulsory middle (3rd) year and senior (4th) year embedded programming courses together with senior year team design (capstone) projects having a major hardware components.

3. WHY EMBEDDED PROGRAMMING INSTRUCTION NEEDS TDD

The above picture demonstrates the extreme skill-level required by the modern embedded expert, and the challenges an instructor faces when trying to introduce students into this world. It is our belief that to be effective, the instructor needs to adopt a number of mechanisms to combat this complexity; we propose that instructors consider adopting TDD as the central component of their armory in this fight. The reasons why TDD works well in this environment are many-fold and are briefly discussed below:

Specification by example: Tradition problem specifications in Computer Science courses are English or structured English statements often augmented with diagrams (as seen in Use Case descriptions and diagrams). While this approach might have merit in large industrial systems, it places significant obstacles in the road of the educational participants. At this point in their career, students struggle to interpret the prose-based approach; often over-inferring and contributing strange meanings to any ambiguous statement. This situation deteriorates further when the students become specification constructers. By contrast, TDD utilizes a simple mechanism which provides students with a much lower entry point in accurately working with requirements. TDD describes requirements simply as a set of acceptance test examples. Each example itself is relatively straight-forward, and can be considered as either an entry in an input/output table or an execution trace through the system.

Automated feedback: Further, using frameworks such as FIT and Fitnesse [7] these test statements are unambiguous and executable providing immediate feedback on whether the system actually "passes" this component of the specification. For most students this is a huge plus; many students struggle with exercises where they cannot crosscheck their answers (e.g. producing designs). While, the community may still be debating the merits of the TDD for large industrial systems; we believe that the approach is ideal for significantly smaller student-oriented projects.

Incremental specification and development: TDD naturally supports the incremental specification; providing the instructor with an excellent mechanism to initially define the core of a system and then, via sets of "extension" exercises, to grow the system enabling the production of large systems; while the students experience a number of small-scale exercises (fortnightly-sized pieces) which initially specify the functional

requirements; and then subsequently add non-functional requirements to produce a semester long project.

Formalized approach to testing: The use of predefined tests in either an acceptance testing or unit testing framework provides a cost-effective mechanism for constructing tests. These tests provide instant feedback and get the student used to the idea that testing is an essential part of the development cycle.

Non-functional specification: Embedded systems often possess non-functional characteristics which rival their functional counterpart in importance. Again, traditional approaches, such as Use Cases, under-represent these components in their requirements documents and notation sets. However, while it is not common practice when using TDD, we believe that writing sets of acceptance tests to articulate the essential non-functional characteristics of a system is straight-forward although not necessarily automate-able. This belief goes outside of the scope of traditional TDD. For example, usability tests are system requirements which can be written as a set of acceptance tests statements that can not be directly evaluated by the system.

Non-functional testing: Again, parting from a standard TDD approach, we have augmented TDD test frameworks to provide mechanisms to test these essential characteristics. Currently, our system provides asserts to test various aspects of execution time or throughput rates and power consumption. We plan to add further assertions to the test framework in the future.

Solid basis for refactoring for non-functional requirements: The functional unit tests provide an essential mechanism for the incremental specification scenario. Here the construction of the functional components is followed by the reconstruction of the system to meet non-functional objectives. The tests provide a simple, but effective, feedback mechanism during the reconstruction process where the students seek to find solutions which will meet the non-functional requirements, while still meeting the functional requirements.

These reasons also provide an initial "window" into other forms of testing (acceptance, regression, stress) that the instructor can use as a "spring-board" to start discussions beyond simple black-box and white-box testing for functional characteristics. Unfortunately, achieving these "advantages" is not straightforward as tool support is a significant problem. To this end, the authors have developed a number of tools which support the utilization of a TDD-approach within an embedded environment.

4. EMBEDDED TDD – INSTRUCTOR'S PERSPECTIVE

4.1 3rd year compulsory embedded-interfacing course experiences

This third year embedded course is in fact the second compulsory embedded course experienced by our students. The first course concentrates on low level machine architecture and teaches the use of assembly code through *SPIM;* an adopted "self-contained simulator that reads and executes MIPS32 assembly language programs (not executables) written for this processor and provides a simple debugger and minimal set of operating system services" [8]. Adding a testing framework to the existing simulator would be a formidable task.

The second course utilizes Blackfin (ADI-BF5XX) based evaluation boards [9] with on-board audio and video capabilities; opening up a wider range of possible embedded experiences. The modern development environment (Visual DSP), combined with JTAG (boundary scan) PC-embedded processor communications, made it possible to extend the course [10] to make use of a formalized approach to testing through a unit test framework *embeddedUnit* [11]. In the early part of this course, an "osmosis" based approach is taken where, during pre-laboratory assignments, the students are provided with testing examples for some aspect of the processor hardware. The students are expected to extend those components when initializing other memory mapped registers. Some of the provided tests, and the "virtual customer discussions" provided in the laboratory hand-out are deliberately designed to fail unless the students take the hints to read the hardware data manual; providing another format for ensuring incremental specification and development.

The practicality of using a formalized embedded testing framework was quickly demonstrated. The first exercise that was given required just 3 lines of C++ code using pointers to initialize 3 memory-mapped hardware registers. However the students identified undocumented behaviour of the Blackfin core timer where registers over-wrote each other; requiring rewriting of the industrial user manuals! This was somewhat to the chagrin of the instructor who had "carefully" TDD-evaluated the exercise by setting registers to the normal operating conditions (same values). This "experience-based" testing procedure totally missed the underlying issues. We have a firm belief that students become better engineers if they can complete "working" components of a full-course project; in a (controlled) environment where they can potentially "smoke" (the less expensive) pieces of equipment. With non-trivial project components (individual laboratories) this often means that the students will spend time outside the laboratory time without instructor and T.A. help in the very "un-user-friendly" C++ / assembly language / hardware environment. TDD has many roles to play here. First, in its regression mode, it provides automatic feedback by allowing the student to replay existing "working scenarios" to identify whether malfunctions are a result of their own code or from damage to the hardware workstation by others. In addition, the acceptance tests, as they incrementally develop, provide control against potential requirements creep. As in the industrial environment, through the tests the student (and professor) can make design decisions about whether a certain project component has value *i.e.* is a basic element needed for future laboratories, or a frill .*i.e.* nice-to-have but not a necessary part of the project. This knowledge makes it easier for the student to balance course loads.

Adopting a TDD approach has considerable advantages for the instructor when modifying the peripherals being interfaced in the laboratory to avoid students easily reusing code solutions from previous years *i.e.* road mapping. The instructor can quickly check that the new laboratory description has not introduced errors, and the scope of the project has not crept so that the students, learning on the job, are unlikely to complete the exercise within the laboratory time slot.

4.2 4th year embedded course experiences

This course attracts students who find a more "hard-core" assembly language approach to embedded systems an asset. This course thus makes use of *embeddedUnit* in an entirely different concept than our 3rd year course. Typically we take an existing technical demonstration provided by the manufacturer and add new features. The students must first demonstrate their understanding the theory behind these new features; an understanding that can be expressed through the testing framework. Then they must ensure that all the required functionality can be performed sufficiently quickly (non-functional specification and testing) *e.g.* in the time between input samples.

First, as can be seen from Fig. 1, the student's develop the design with embedded-specific, but un-optimized, high level C++ code using the TDD framework. Although moving to optimized C++ code is achievable by setting a single development environment variable, the students must become intimately aware of the consequences at the register level. In Fig. 1, it is seen that the same tests are used for both optimized and un-optimized code segments; combined with non-functional timing tests. If the optimized C++ does not meet the specified performance tests, the student is then expected to optimize the code at the architectural level via customized assembly code. To fully utilize all the tools available to identify the offending code sessions requires knowledge of processor specific hardware based performance registers, or through profiling using simulators or statistical profiling of the running system. However, there is a steep student learning curve associated with each of these sophisticated tools. Given that the opportunity to actually use some of the tools will be infrequent during the duration of the course, we have taken the approach of having groups of students demonstrate their experiences to the remainder of the class as part of the elective small-group component of the course.

4.3 Final year capstone project experiences

At the research level we have been successful in prototyping embedded-*FIT* [7] — an acceptance test framework for embedded systems. However, neither the average student nor the average embedded industrial customer possesses sufficient experience with the concept to allow a full deployment of the methodology for true specification by example. Even with these difficulties, we have found application of the TDD process useful when acting as a moderator between students and sponsor. While a hardcopy document may result, rather than the desired executable specification document, the instructor can recognize whether iterative (incremental), customer centric, discussions have occurred. This year, given that the students have not been previously exposed to embedded TDD, we have required that the team demonstrate use of an *embeddedUnit* [11] testing framework as one milestone within the project (faculty advisory marks). It will be interesting see whether this required milestone affects the ability of the team to express their design ideas to complete their electively chosen projects.

5. STUDENT PERSPECTIVE

A simple comparison between the 2005 and our proposed use in 2007 [10] of *embeddedUnit* in our 4th year elective class shows that the whole testing framework is under flux as we better understand the role that TDD can play within the embedded environment, both industrially and educationally. For example, we have recently recognized how to use the Blackfin data and instruction watch registers to provide a hardware-based, "virtual instrumentation" approach to identify data-races in multi-threaded embedded systems with an efficiency of 20 to 200 times that of current software instrumentation approaches. Adopting this capability within a TDD framework for our existing 3rd year RTOS course will have dramatic consequences. In addition, the new capability is requiring a significant rethinking of the syntax

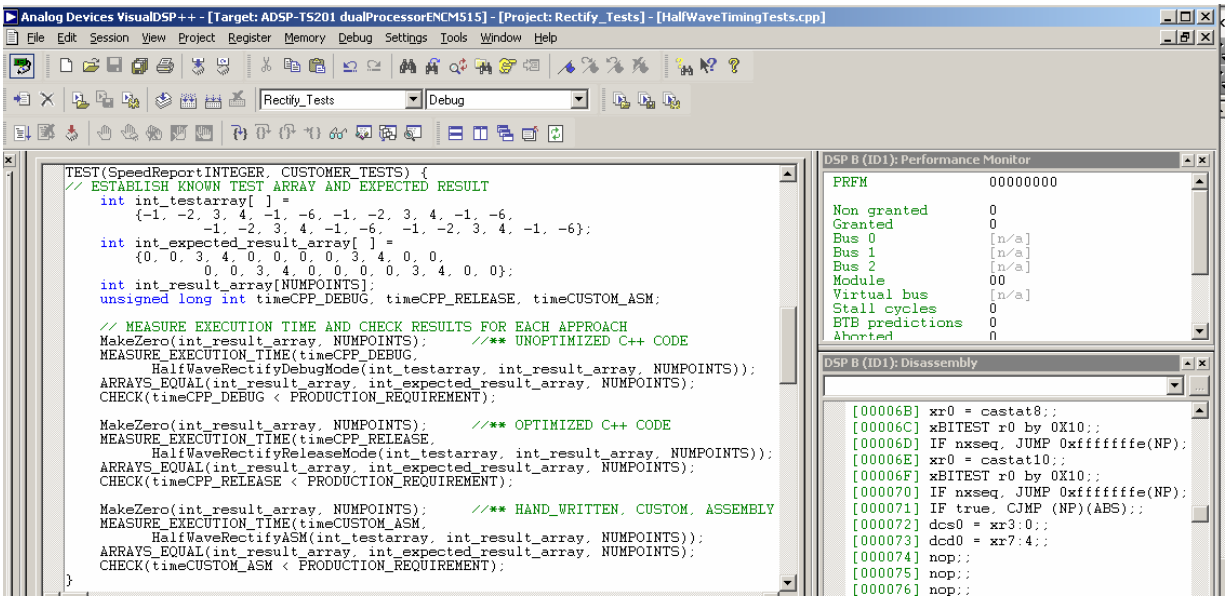

Fig. 1. Tests are shown that illustrated the use of the dual processor ADI-TS201 TigerSHARC *embeddedUnit* (E-TDD) environment [11]. Note that TDD will be used in a standard "explore the design" manner for the un-optimized C++ code, but is transformed to a more "design by contract meets TDD" technique for the "less debuggable" optimized C++ and assembly code versions.

appropriate for embedded testing. Given that our testing framework has yet to remain stable for even the duration of one half course, the time has not yet arrived for a formal student-based study of the effectiveness of *embeddedUnit*, and TDD in general, within the embedded educational scene. However, we can offer the following comments; some of which are antidotal, while others have more precise metrics seen through the performance of students in assignments, quizzes, laboratories and exams.

- For some students TDD is just another cross they have to bear. By contrast other students have expressed the fact that this is the first formalized testing approach they have seen, and wonder how other professors can be persuaded to adopt such practices.

- As mentioned earlier, the students appreciate the ability to use existing tests to check that equipment is working outside of regular laboratory hours. However, for many that ability stays at the "I know the equipment is not working, lets get the T.A. to fix it" level rather than allowing solution solving to occur. However, that alone has considerable improved the student attitude to these laboratory based courses which are perceived as "extremely time consuming" -- a comment repeated many times across 15 years of student surveys.

- At the 4th year level, as discussed earlier, the students can be made aware of the theoretical and practical aspects of the telecommunications and image processing code they are developing through TDD. When moving to optimized assembly code, it is very easy for the student to become totally embroiled in issues surrounding the efficiency of coding. However, the availability of customer requirements, easily understandable in the form of tests, does enable the big picture to be quickly re-established provided a little effort is taken by both the student and the instructor.

We have found that the TDD framework useful for exploring the student's understanding of design concepts during final exams. But we do issue a word of caution. The first year we tried this, many of the good students "knew" what was expected of them for the black box testing of the embedded systems *e.g.* coding hardware interrupt service routines so that they could be tested via the raising of software interrupt flags. However, because of the wording of the questions, we also had to award full marks for answers that effectively only demonstrated the student's understanding of how to use the TDD syntax to repeat the requirements stated in the question. However, this useful acquired skill that had already been tested else where in the final exam. This year we were more precise in the wording for such questions, and saw a definite split between the students who had a thorough understanding of the design / testing process and those who seem to have more heavily relied on their laboratory partner during testing.

6. CONCLUSION

In this paper, we have demonstrated that modern embedded system construction is a highly complex undertaking; and that the instructor must deploy mechanisms to "melt away" this complexity and allow students to develop essential skills. It is argued that test driven development (TDD) is a highly appropriate mechanism to achieve this multi-faceted objective. TDD supports the subdivision of the complexity by a number of mechanisms: specification by example, non-functional specification, incremental specification, incremental development, formalized approaches to testing, automated feedback, non-functional testing, and a solid basis for refactoring for non-functional requirements. Further, the paper outlines our experiences in deploying the approach; and while, we would characterize this as a work still in progress; the feedback from the student body is encouraging.

7. ACKNOWLEDGEMENTS

Financial support provided by the Province of Alberta, Canada, Analog Devices (Canada) Ltd and the Natural Sciences and Engineering Research Council (NSERC), Canada.

8. REFERENCES

[1] Greene, W. Agile Methods Applied to Embedded Firmware Development, *Agile Development Conference*, 71 – 77, 2004.

[2] Dahlby, D., *Applying Agile Methods to Embedded Systems Development*, embuild.org/dahlby/agileEm/agileEm.doc.

[3] Edwards, S.H., Rethinking Computer Science Education from a "test-first" Perspective, *OOPSLA, 2003*.

[4] Marrero, W; Settle, A., Testing First: Emphasizing Testing in Early Computer Science Courses, *ITICSE, 2005*.

[5] Beck, K., *Test-Driven Development: By example*, Addison-Wesley, 2002.

[6] Mugridge, R.; Ward, C., *Fit for developing software: framework for integrated tests*, Prentice Hall 2006.

[7] Chen, J., Smith, M., Geras, A., Miller, J., Ko, L., Making Fit / FitNesse Appropriate for Biomedical Engineering Research, *Proceedings of 7th International Conference on eXtreme Programming and Agile Processes in Software*. Oulu, Finland, Oulu, Finland. 186-190.

[8] Larus, J, *SPIM: a MIPS32 Simulator*, www.cs.wisc.edu/~larus/spim.html, accessed Dec 25, 2006.

[9] Analog Devices, *Blackfin Evaluation board*, www.analog.com/processors/platforms/index.html, accessed Dec. 25, 2006.

[10] Smith, M., *Assembly language and Interfacing*, www2.enel.ucalgary.ca/People/Smith/ accessed Dec. 25, 2006.

[11] Smith, M., Kwan, A., Martin, A., Miller, J., E-TDD – Embedded Test Driven Development: A Tool for Hardware-Software Co-design, *Proceedings of 6th International Conference on eXtreme Programming and Agile Processes in Software, Sheffield, UK*, 145 – 153, 2000