ON THE BENEFITS OF INTEGRATING SYSTEMATIC VERIFICATION INTO CS1 AND CS2

Peter Walsh

Department of Computing Science

Malaspina University College,

Nanaimo, B.C., V9R 5S5 Canada

Jim Uhl

Department of Computing Science

Malaspina University College,

Nanaimo, B.C., V9R 5S5 Canada

Abstract

This work was motivated by two problems we experienced while teaching the introductory computer programming courses CS1 and CS2. Firstly, the quality of code written by our students was poor and showed little improvement over time. The norm was overly complex code that exhibited about 80% of the specified functionality. Secondly, it was difficult and labour intensive to evaluate student code effectively. To address these problems, we integrated systematic verification into both our CS1 and CS2 courses. We found that systematic verification facilitated the key engineering principles of implementation to specification and quality evaluation. We observed an increase in the quality of student code and a decrease in the effort required to evaluate student code.

1 INTRODUCTION

We have over ten years experience teaching introductory computer programming at the CS1 and CS2 levels. While teaching these courses, we observed student code that was overly complex and exhibited about 80% of the specified functionality. In addition, the human effort required to evaluate student code was substantial. Often, insufficient resources resulted in ineffective feedback to the students concerning their code. This served to magnify the problems with quality and evaluation.

We view systematic verification as two complementary methods of verification *viz.*, testing and inspection (see section 2) [1, 2, 3]. Our first experience with systematic verification was in a senior undergraduate course in software engineering. For many of our students, it was the first time that their programs were thoroughly tested and inspected based on a set of criteria. They had previously excelled in the *ad. hoc.* evaluation atmosphere based on loose specifications and little or no testing. In stark contrast, many struggled with the notion of implementation to specification and systematic testing. This, coupled with the problems in quality and evaluation mentioned previously, led us to conclude that systematic verification should be introduced earlier in our program. To this end, we integrated systematic verification into our CS1 and CS2 courses.

All our CS1 and CS2 programming exercises are issued with a specification, inspection criteria, and a suite of basic acceptance tests (BATs). We test and inspect student code relative to the specification and inspection criteria to determine code quality. The testing process is automated. Students can run the BATs as many times as they wish before submitting their work to the instructor. No credit is given for work unless it passes the BATs. In addition, the instructor runs a comprehensive test suite called the instructor's acceptance tests (IATs) to further establish quality.

BATs and IATs are used systematically to expose an implementation that does not meet its specification. From the student's perspective, they reinforce the key engineering principle of implementation to specification. From an instructor's perspective, they establish confidence (or otherwise) in student code. Available resources can then concentrate on issues of style, architecture and efficiency. In our experience, the net result is better written code by the student and a more effective evaluation by the instructor.

Section 2 describes our approach to inspection. Section 3 describes our experience with CS1 and details our approach to testing CS1 programs. Section 4 describes our experience with CS2 and details our approach to testing CS2 components. Section 5 discusses the costs and benefits of a BATs/IATs/Inspection strategy and section 6 contains concluding remarks.

2 INSPECTION

Software inspections are based on peer review of design and implementation work products. Studies suggest that inspection is a key component of effective software engineering practice [4]. We believe it is important to impress upon our students the need for inspection by evaluating their programming exercise submissions based on a set of criteria published with exercise specifications. In general, inspection is used to evaluate those qualities of the code not concerned with run-time correctness.

Industrial inspections concentrate on a variety of concerns, from checking for style violations to proving that all loops terminate. Our inspection criteria are based on these ideas, but modified to account for the fact that we are inspecting the work of programmers just learning their craft, rather than professional programmers. The goal is to give our students an opportunity to explore design and implementation alternatives. Over time, we add criteria to help move them towards writing understandable, robust, reasonably efficient, programs. On a more pragmatic level, the criteria must allow grading to be completed in a reasonable amount of time.

Figure 1 shows a sample of the inspection criteria we use for both our CS1 and CS2 courses. For the most part, we adopt the coding and commenting conventions suggested in the course textbooks [5, 6]. The overriding concern is one of simplicity, which is a key concern of maintainable code. Thus, comments should be neither more nor less than required. Code should not be duplicated. The decomposition should be clear and understandable. Together, these criteria help direct the student toward good programming habits.

3 COMPUTER PROGRAMMING I (CS1)

CSCI 160 is a first course in computer science offered annually at Malaspina University-College. Topics include structured programming, top-down program design, functions, recursion, and an introduction to dynamic data structures (CS1 in the ACM curriculum). The course extends over fourteen weeks with four hours of lectures per week and has a maximum enrollment of 128 students.

We teach C++ in a Unix environment using FUSE, DEC's interactive development environment (IDE) [7]. Programming exercises are issued with a specification, inspection criteria

- Coding Style
  - Is the code indented properly? Are newlines in the correct places?
  - Do identifiers follow the prescribed convention?
- Comments
  - Can a function be re-constructed from its comments alone?
  - Are the comments free from implementation details?
  - Is duplication of documentation avoided using references?
- Semantic issues
  - Are expressions/statements/functions/classes/etc. as simple as possible?
  - Are constants named, rather than appearing as "magic numbers" in the code?
  - Is code effectively abstracted to remove duplication?
  - Is all the required functionality present? E.g., is all dynamically allocated memory freed?
- Component Structure
  - Is the submission (program/ADT) well-structured?
  - Do the components and their interfaces make sense? Are they reasonably general?
  - Are parameter passing conventions used correctly?
  - Are there any global variables?

Figure 1: CS1 and CS2 Sample Inspection Criteria

and BATs. A typical assignment is to write a complete interactive program or augment an existing interactive program. For example, Appendix A is a specification for a palindrome assignment we used in the last offering of the course. All CSCI 160 specifications contain a description of the problem, sample output, and a section on assumptions/restrictions.

The CS1 testing scheme centers around the manipulation of three directories called `tinp`, `texp` and `tact` [1]. Each file in `tinp` contains a test input. Each file in `texp` contains the output we expect to get when the program-under-test is executed with the test input from the corresponding file in `tinp`. Each file in `tact` contains the actual output the program-under-test produces using the test input contained in the corresponding file in `tinp`. BATs and IATs are constructed by adding files with test input to `tinp` and files with expected output to `texp`. The program-under-test fails a test case when actual behaviour deviates from expected behaviour. We check for this using Unix's `diff` command. BATs and IATs can be executed through FUSE or from the command line.

Student exercise submissions are evaluated based upon testing and inspection. Ten points are allocated to testing and ten points to inspection for a maximum of twenty points. Student submissions that fail the BATs receive zero points in total. The remaining submissions receive five points for having passed the BATs, and an additional five points if they pass the IATs. Each submission is printed and inspected. Violations of the inspection criteria (see section 2) are noted and points deducted, starting at a maximum value of ten inspection points. The number of points deducted for a particular violation may shift over the duration of the course to eliminate poor coding practices. The result of this process is an *Instructor Inspection Report* indicating precisely where the student lost points and why.

Figure 2 shows a graph that plots final exam scores against the combined Exercise 5 and 6 scores for each student (data from CSCI 160, Fall 1998). Developing correct solutions to these exercises required a good understanding of the material in the course. As can be seen, performance on these exercises was a good predictor of performance on the final exam. Two anomalies are worth noting. Two students did very well on the exercises, but failed the final exam: we strongly suspect these individuals submitted other people's solutions, and thus were unable to answer similar exam questions. A cluster of students scored roughly 40% on the final exam, but scored about 75% on the exercises: these people either failed the IATs or lost

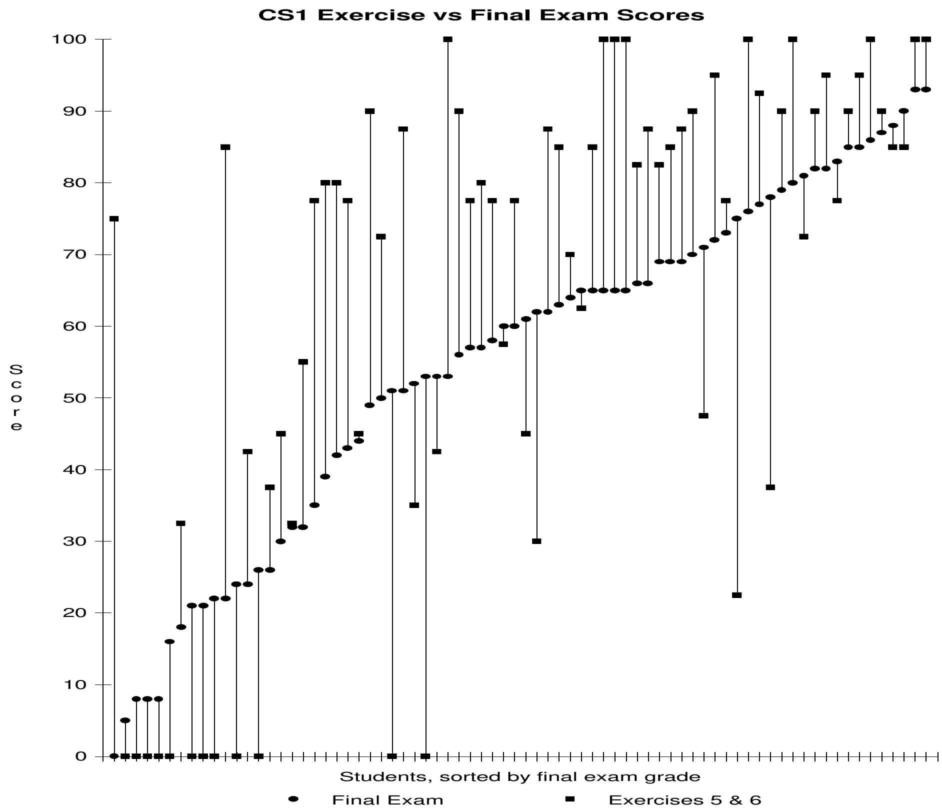

Figure 2: CS2 Exercise vs Final Exam Scores

points due to inspection, and thus were still struggling with the concepts at the end of the course. These results are not surprising because the concepts in introductory programming are additive: failing to learn the early concepts all but precludes learning subsequent concepts, and it is necessary to have a grasp of all the concepts in the course before proceeding to Computer Programming II.

4 COMPUTER PROGRAMMING II (CS2)

CSCI 161 is a second course in computer science offered annually at Malaspina University-College. Topics concentrate on the design, implementation and use of abstract data types (ADTs) (CS2 in the ACM curriculum). The course extends over fourteen weeks with four hours of lectures per week and has a maximum enrollment of 64 students.

We teach C++ in a Unix environment, taking advantage of features suited to constructing ADTs, including classes and templates. Student programming exercises are issued with a specification, inspection criteria and BATs. Early programming exercises have the students implement an ADT to specification. Later exercises use previous ADTs to construct new ADTs, as well as the development of complete programs that use various ADTs.

ADT implementation exercises are given in the form of an interface (.h) file that includes a class definition, and documentation for all the functions associated with the ADT (see Appendix B for a sample). The students play the role of implementor, and thus are allowed to change the private part of the class and to add in-line function definitions as they see fit. The public part of the class visible outside the class, however, is strictly off-limits and this is checked during inspection.

The CS2 testing scheme is based on *cew*: a C++ component exerciser workbench [8]. A *cew* script consists of a C++ program embedded with test cases that allow ADT operations to be invoked, and their return values and signaled exceptions to be checked. The component-under-test (CUT) fails a test case if actual behaviour deviates from expected behaviour.

Appendix B contains an example *cew* script for `IntSet` (see `IntSet.script`). The script contains two kinds of test cases, *viz.,* `cew_Ncase` and `cew_Ecase`. A `cew_Ncase` is for testing normal behaviour and its syntax is *cew_Ncase(trace, actval, expval)*. A *trace* is any manipulation of the CUT through its public interface. The *actval* is an expression that is evaluated after the trace. Its value is taken as the "actual value" of the trace. *expval* is the value that *actval* is expected to have. *cew* reports an error if *actval* is not equal to *expval*. A `cew_Ecase` is for testing exceptional behaviour and its syntax is *cew_Ecase(trace, expexc)*. *expexc* is the exception the *trace* is expected to throw. *cew* reports an error if no exception is thrown or if an exception other than *expexc* is thrown. *cew* also allows for the examination of ADT state if a test case fails. This feature can be used to debug both CUT code and test cases.

The mechanism used to evaluate student exercise submissions mirrors the mechanism we use in our CS1 course. The inspection criteria move towards verifying effective use of abstraction and the re-use of code.

Figure 3 show a graph that plots final exam scores against the aggregate exercise scores for each student, both normalized to 100 points (data from CSCI 161, Winter 1999). The exercise scores were not as good a predictor of final exam performance as in the CS1 case. Note, however, that only one student scored over 75% on the exercises (*i.e.,* consistently passed the IATs) scored less than 50% on the final exam. On the other hand, of those who passed the final exam, several students not only scored less than 75% on the exercises, but less than 50%(!). Interestingly, most of the students in this category had taken the initial course from instructors who do not use our approach. In many of these cases, it took over half the exercises for the student to understand the importance of passing the BATs (consistently receiving zero points did not seem to concern most of them). Finally, it should be noted that while CSCI 161 teaches additional programming features of C++, its primary goal is to teach abstract data types, many of which are relatively independent. Thus, it is possible for the student to pass the final exam knowing only part of the material.

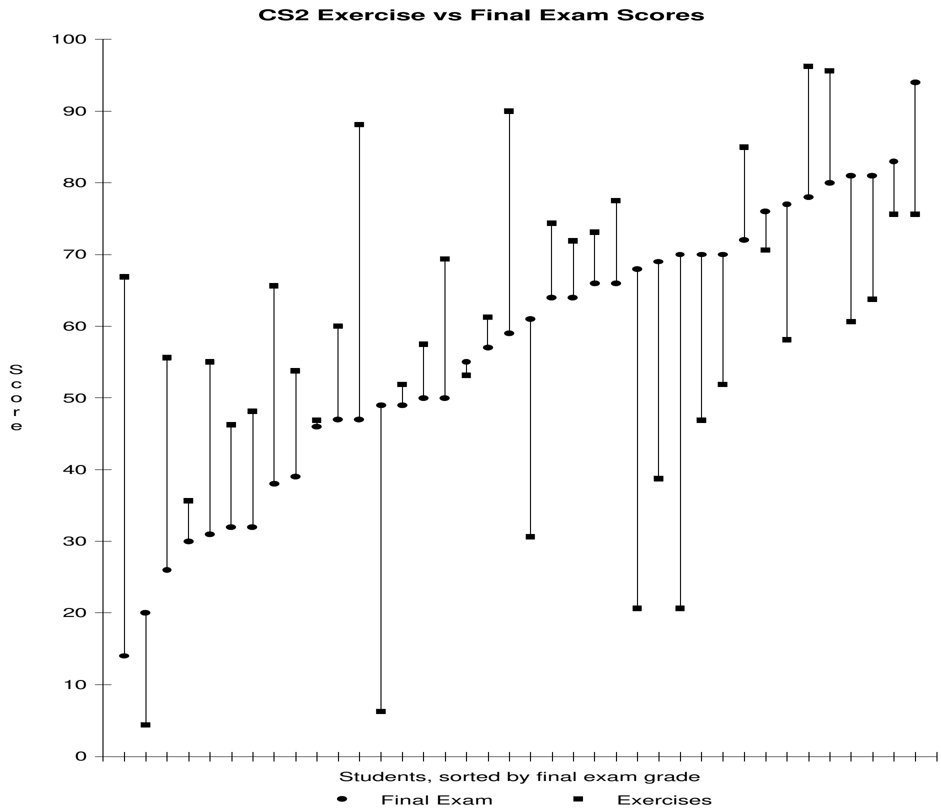

Figure 3: CS2 Exercise vs Final Exam Scores

From the instructor's point of view, the testing and inspection approach has significant benefits at the CS2 level. ADTs are very amenable to automated testing. Effective inspection begins to unify the coding style adopted by the students, resulting in code of high quality that is easy to read and debug.

5 SYSTEMATIC VERIFICATION (COST BENEFIT ANALYSIS)

The use of systematic verification requires a significant time investment up-front and yields a sizable downstream benefit to both student and instructor. Non-recurring start-up costs include the development of an automated testing infrastructure that supports the development and execution of:

- BATs on interactive programs for CS1 students
- BATs on ADTs for CS2 students
- BATs and IATs for an entire batch of student exercise submissions for the instructors of both CS1 and CS2

Once in place, these components can be re-used at little or no cost, allowing time to be used more productively on tasks that cannot be easily automated. As of this writing, we have the above infrastructure in the form of UNIX shell scripts for CS1 and *cew* for CS2. In our experience, *cew* is quite effective both from the perspective of the instructor developing BATs and IATs and the student adding test cases to the BATs to verify the behaviour of their code. On the other hand, the testing used in CS1 is both rigid and difficult to understand for CS1 students. They often become frustrated when they fail a test case because their output differs only cosmetically from the expected output. Moreover, because the current CS1 testing infrastructure is cumbersome, students find it difficult to add their own test cases.

For systematic testing to be feasible, the programming exercises must be specified fully and the BATs developed before the exercise is given to the students. Typically, the instructor also develops a solution to the problem at the same time. As above, these are up-front costs and are partially recurring if one changes the specifications each time the course is offered.

From the instructor's perspective, the use of systematic verification (as we have described it) has the the following benefits:

- reduced interaction with students with regard to exercise functionality
- automatic application of BATs and IATs
- uniform evaluation of student code based on the BATs/IATs and inspection criteria
- reduced effort to identify complicated or poorly structured code (once quality is established with BATs/IATs)
- reduction in the number of grade appeals and the time to deal with them
- intellectual satisfaction from developing precise specifications, BATs, IATs and solutions

From the student's perspective, the use of systematic verification has the the following benefits:

- reduced interaction with the instructor with regard to exercise functionality
- reduced likelihood of exercise requirements-creep
- fair evaluation with respect to the other students in the course
- clear and concise feedback regarding inspection criteria
- reduced uncertainty with regard to grading criteria
- availability of instructor's solution immediately after the exercise due date
- intellectual satisfaction from successfully passing the BATs and the IATs

6 CONCLUSION

Acceptance testing is widely used in the software industry where programs are not accepted by quality-control departments until they pass basic acceptance tests (BATs). Only then are they subjected to a comprehensive set of acceptance tests (IATs). By using BATs and IATs, we have brought a real-world feel to our CS1 and CS2 courses.

The BATs/IATs/Inspection strategy has proved effective in addressing the problems of code quality and code evaluation. Over the course of CS1 and CS2, we observed continued improvement in code quality measured as a percentage of the number of student submissions that passed both the BATs and the IATs. We also observed a continued improvement in coding style. We attributed this to the clear feedback students received in the form of the instructor's inspection reports. Detailed inspections of the first few exercises eliminated many questionable coding practices. In particular, poorly indented code and useless comments all but disappeared.

We had hoped that exposure to systematic verification at the CS1 and CS2 levels would be evident in the quality of code developed by students for more advanced courses. This was not the case. In courses where we did not employ systematic verification, we observed a quick regression to poor programming practices that we thought had been eliminated in the CS1 and CS2 courses. This led us to the key insight that there is no compliance without enforcement. Consequently, we now employ systematic verification in our assembly language programming course (CSCI 261 using 80386 assembly language), logic design course (CSCI 355 using VHDL) and our two software engineering courses (CSCI 265 and CSCI 365 using C++). Our experience with these courses mirrors our experience with our CS1 and CS2 courses.

APPENDIX A

```
Description
===========
Write a C++ program that will read a sequence of characters from stdin
and determine if the sequence is a palindrome.  Your palindrome test
should be case insensitive (that is, treat lower case letters the same
as upper case letters) and should ignore non alpha-numeric characters.
Note that a palindrome is a sequence of characters that reads the same
backwards as it does forwards.  See the Sample Output below for some
examples.

Sample Output 1
===============
Enter the sequence of characters:
Nurses run
The sequence is a palindrome.

Sample Output 2
===============
Enter the sequence of characters:
Nurses ru
The sequence is NOT a palindrome.

Assumptions/Restriction
========================
You must read the sequence of characters from stdin using the get
function (see page 95 of the text).  The end of input is signaled by
"end of file," which may be tested using cin as a boolean (see pages
146-147 of the text).  You should only store a maximum of 79
alpha-numeric characters from the sequence.  You may NOT use any
string library.  You must use the ctype library for character
processing (see pages 255 and 318 in text).  Note that the empty
sequence, that is a sequence of zero alpha-numeric characters, is
considered a palindrome.

Test Set
========
see ~csci160/Pex5/tinp
```

APPENDIX B

```
------------------------------ IntSet.h ------------------------------
// FILE: IntSet.h
// CLASS PROVIDED: IntSet (a container class for a set of integers)
//
// EXCEPTION CLASSES
//   class FullExc
//   class DuplicateExc
//
// MEMBER CONSTANTS for the IntSet class:
//   static const int CAPACITY = _____
//     IntSet::CAPACITY is the maximum number of items that an IntSet can hold.
//
// CONSTRUCTOR for the IntSet class:
//   IntSet( )
//     Postcondition: The IntSet has been initialized as the empty set.
//
// MODIFICATION MEMBER FUNCTIONS for the IntSet class:
//   void add(int entry)
//     Postcondition: If size() = CAPACITY, throw FullExc; if entry
//     is already member, throw DuplicateExc; otherwise, add entry to
//     the set.
//
// CONSTANT MEMBER FUNCTIONS for the IntSet class:
//   int size( ) const
//     Postcondition: Returns the total number of items in the set.
//
//   bool is_member(int target) const
//     Postcondition: Returns true if and only if target is a member
//     of the set.
//
// VALUE SEMANTICS for the IntSet class:
//    Assignments and the copy constructor may be used with IntSet objects.

#ifndef INTSET_H
#define INTSET_H

class FullExc      {};
class DuplicateExc {};

class IntSet {
public:
    const int CAPACITY = 3;

    IntSet( );
    void add(int);
    int size( ) const;
    bool is_member(int) const;
```

```
private:
    //**************************************************
    // YOU ARE RESPONSIBLE FOR COMPLETING THIS SECTION *
    //**************************************************
};
#endif /* INTSET_H */
------------------------------ IntSet.script -------------------------------
#include <iostream.h>
#include "IntSet.h"

include(cew.c++)

cew_Start_Menu
   cew_Menu_Item(a,
 Add to s,
 int x;cout << "Enter Integer: "; cin >> x; s.add(x))

   cew_Menu_Item(d,
 Delete from s,
 cout << "Not available" << endl)
cew_Stop_Menu

cew_Start_Exception_Handler_Builder
   cew_Build_Handler(DuplicateExc)
   cew_Build_Handler(FullExc)
cew_Stop_Exception_Handler_Builder

int main()
{
    cew_Set_Mode(cew_Interactive_On_Failure)

    { IntSet s; cew_Ncase(, s.size(), 0) }
    { IntSet s; cew_Ncase(, s.is_member(1), false) }
    { IntSet s; cew_Ncase(s.add(1), s.size(), 1) }
    { IntSet s; cew_Ncase(s.add(1), s.is_member(1), true) }
    { IntSet s; cew_Ecase(s.add(1); s.add(1), DuplicateExc) }

    cew_Summary

}
```

References

[1] D. Hoffman and P. Strooper, *Software Design, Automated Testing, and Maintenance*, ITP, 1995.

[2] D. Hoffman and P. Strooper and P. Walsh, "Teaching and Testing", *9th Conference on Software Engineering Education*, Daytona Beach Florida, 1996.

[3] D. Hoffman and P. Strooper and P. Walsh, "Teaching Software Design and Maintenance: A Document-Driven Approach", *1996 Western Canadian Conference on Computer Education*, Prince George, B.C. Canada, (10 pages) May 2nd 1996.

[4] M. Van Emden, "Structured Inspections of Code", *Software Testing, Verification and Reliability*, Vol 2, 133-153, 1992.

[5] N. Dale and C. Weems and M. Headington, *Programming in C++*, Jones and Bartlett, 1998.

[6] M. Main and W. Savitch, *Data Structures and Other Objects Using C++* , Addison Wesley, 1998.

[7] Compaq (Digital Products and Services), *DEC FUSE*, `http://www.digital.com/info/fuse/prodinfo.htm`

[8] P. Walsh and J Uhl, "*cew*: Component Exerciser Workbench", *Technical Report MC10*, Department of Computing Science, Malaspina University-College (in preparation) 1999.