Automatic correction of C programming exercises through Unit-Testing and Aspect-Programming

Andrea STERBINI
Dipartimento di Informatica, Università di Roma "La Sapienza"
Rome, I-00198, Italy
sterbini@dsi.uniroma1.it

and

Marco TEMPERINI
Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza"
Rome, I-00198, Italy
marte@dis.uniroma1.it

1. ABSTRACT

We present a framework for the automatic testing of C Programming assignments that is based on Unit tests and Aspect programming. Students are required to implement a set of C functions and are given each function's prototype. Teachers are required to write unit tests for each function, a "reference implementation" of each function and a feedback message for each test failed. The system applies the unit tests to the code and replaces a failing function with the corresponding reference implementation to test also functions depending on the failed one.

We are developing the system while testing it on the homeworks submitted for two courses: "Programming 2" and "Programming Laboratory", delivered during the second term of first year of our Computer Science MS degree. We present the design of the system and the issues solved so far to test 4 homeworks (a total of 186 tests run over 28 implemented functions).

Keywords: Automatic grading, automatic correction, software testing.

2. MOTIVATION

In sciences and engineering faculties, such as the ones we belong, computer science foundations and programming are topics normally thought in one or more courses, during the first year of every study programmes. Such courses are normally populated by 90 to 150 students, and managed by one or two teachers (one "in charge" and the other as a support). In our experience, it is not infrequent to see part of the students disattending the practical activities assigned in homeworks and laboratory, and then trying the exam without proper training in writing, running and debugging programs. The environment we are confronting at present is that of a couple of courses, joined by (the same) 130 students each ("Programming 2" and the attached "Programming Laboratory", starting march first 2004 and spanning ten weeks). In each course we are giving homeworks every two weeks, both to analyse their progress, and to give them useful feedback through the homework correction. While we see that such activity could strengthen the students' motivation and satisfaction in programming (not to mention increasing their performance as a final result), we see the effort needed on our side as well: an overall number of 650 student's homeworks to be corrected for each course. This can be managed only by using a framework for automatic correction and feedback generation.

So far (on previous courses) we have been experimenting, a bit roughly, with black-box-testing of the student program, by checking on several input sets the conformance of its output with the expected ones. This approach is helpful, though in some cases - let's say 15% of them - we had to go over a direct hand correction of the code to cope with very simple errors. Such an approach is too simple minded indeed and could lower the student's motivations. E.g. when "insignificant" differences between the expected output and the student's one let the whole test fail: as normally the students are more focused on the logic of the exercise, than on its I/O functions. Providing the student with a whole driver containing the proper I/O functions, would heal some of these errors, yet we think this would move the frontier of the testable problems just a little further. Is it possible to provide the teacher with a sufficiently simple framework to support the definition of exercises, the specification of the various detailed tests, and their final automated correction?

In the development of a framework to answer to the above question we have the following goals:

- make the correction completely automatic. This way the corrector can be used also as a self-evaluation tool in distance-learning settings, by leaving the students test their code and resubmit it, in an iterative improvement process;
- give the student a detailed correction report that explains, where possible, what the error was and how to avoid it;
- avoid the propagation of errors that lead further tests to failure in the same exercise because of functional dependencies;

- collect evidence of side-effect related errors (e.g. improper use of global variables or of references or pointers)

3. RELATED WORK

A similar problem has been solved by Morris [1] for programming courses of Java. In his work he uses Java reflection to analyse the student code and to replace failing functions with his reference implementation.

In C a similar technique is harder to apply because reflection is not available. Yet, we have found a viable alternative to reflection in Aspect Programming [2]. Aspect Programming allows the definition of "aspects", i.e. the specification of code that should be interspersed in the program to add some functionality (e.g. counting the number of calls of a given function). With Aspect programming we can easily instrument the student code so to:

- test each function against a set of unit-tests
- replace failing functions with a reference implementation
- count function calls to check algorithm complexity

Tests are written by following the software engineering Unit-test methodology, where the smallest possible tests (unit tests) are defined and then collected in test suites.

4. DESCRIPTION OF OUR FRAMEWORK

Data sources

The specification of a programming exercise is made of the following items produced by the teacher:

- the exercise description, listing the function prototypes that the student should implement,
- the unit tests needed to test all kinds of errors we want to detect,
- the reference implementation of each function,
- the feedback comments to be attached to each test,
- the dependencies among functions, to properly order tests so that errors are not propagated to depending functions

All the above items (except the exercise text) are used to correct the student's implementation of the required functions (solution)

Automatic correction steps

The correction process follows these steps:

- we instrument the student's code with the test code,
- we run tests, ordered depending on the functional dependencies between functions,
- failing functions are replaced with their reference implementation to avoid that earlier errors are propagated to depending functions
- we collect the feedback from tests to tell to the student what errors have been discovered. Feedback is sent by email to the student or published on a web site.

Further opportunities

The above steps implement an adequate automatic correction tool for a generic programming course. In our specific course the student are taught software engineering techniques and should implement the tests needed to check their code correctness. Thus, beside the above sources, we should manage and correct the student's tests.

To correct the each student's tests we apply to all student's solutions the following steps:

- use the student's tests against all the other student's programs,
- store the test's answers,
- compare the student tests answers with the teacher test answers,
- tests are considered 'correct' when they behave in the same way than the teacher's ones
- when tests behave differently we produce a report showing the different behaviour observed

Writing and reusing tests

We use unit-testing as a method for writing tests. We have chosen the cppunit [3] open-source implementation of the unit-test framework for C and C++. Tests can be written both as single functions or as methods of test classes. The tool is easy to use and allows also the detection of the exceptions thrown by the tested code. As test development is a time consuming activity, test generalization and reuse is a key issue that will distribute the energies spent for writing tests over their reuse in several similar exercises. Good reusability and generalization of tests is obtained both by using C++ templates to write tests that are parametric on the types of the used data-structures, and by using a hierarchy of test classes that can be extended to produce more specific tests.

Catching evidence of subtle errors

By testing separately each implemented function we catch only the errors that were expected by the teacher. Subtler errors happens when the student uses global variables or side-effects to communicate informations between different parts of the program. To catch such interactions (or at least to obtain a symptom of their presence) we want to run tests on more than one student function at the same time. I.e. a subset of m student functions are embedded in the reference implementation of the whole program. For an exercise where N functions have been implemented, a worst case of 2^N possible tests should be run to examine all subsets of the N functions. Exposing the student to such amount of output is useless, thus we filter the test's output and keep only two notable cases:

- when we test a subset M of m student functions and:
  - the tests of a function are PASSED but at least one of the m tests already run on the subset M x were FAILED,
  - or the tests of a function are FAILED but all the m tests already run on the subset M x were PASSED.

5. A TOY EXAMPLE OF FUNCTIONS, TESTS AND ASPECTS

Functions, tests and reference implementations

```
#include <stdio.h>
#include "main.h"

int square(int N) {     // WRONG implementation
        return N*(N-1);        }
int square_OK(int N) {  // REFERENCE implementation
        return N*N; }
bool test_square()    { // TEST
        int i = -17;
        return (i*i == square(i)); }

int f(int N) {          // CORRECT implementation
        return square(N) + 2; }
int f_OK(int N) {       // REFERENCE implementation
        return square(N) + 2; }
bool test_f() {         // TEST
        int i = -18;
        return (i*i+2 == f(i)); }

int main() {
        printf("f(4) = %d (18)\n", f(4));
        printf("square(4) = %d (16)\n", square(4));
        return 0; }
```

Figure 1: Toy example of code to be instrumented.

In figure 1 we show a toy example that shows how aspects can be defined to support our framework. First we define a pair of functions ('f' and 'square') with their reference implementations ('f_OK' and 'square_OK') and their tests ('test_f' and 'test_square').

6. A SIMPLE ASPECT DEFINITION

In figure 3 we show a sample aspect definition used to instrument the above program. The aspect instruments the code both to execute the corresponding tests ('test_f' and 'test_square') and to replace failing functions with the corresponding reference implementations ('f_OK' and 'square_OK').

When the aspect is applied to the code we obtain a program that produces the output shown in figure 2. From the example of execution (when 'f' is correct and 'square' is not) we notice how:

- the tests are run in the proper order
- the 'square' function is replaced both during 'f' test and during the program execution
- the 'f' function is found correct and is not replaced

```
testing the 'square' function ...  FAILED!
testing the 'f' function ...
        using REPLACED version of int square(int)
        PASSED!
computing f(4) ...
        using ORIGINAL version of int f(int)
        using REPLACED version of int square(int)
        f(4) = 18 (expecting 18)
computing square(4) ...
        using REPLACED version of int square(int)
        square(4) = 16 (expecting 16)
```

Figure 2: Output of the execution of the instrumented code.

7. STATUS OF THE PROJECT

The project has begun two months ago as a pilot project implemented by a Master thesis student. We have written and run tests for two courses: "Programming 2" and "Programming Laboratory", delivered to freshmens during the second term of the first year of our Computer Science MS degree. The first course teaches programming in C with dynamic data structures (lists, trees, graphs), while the second is a laboratory on software engineering practices, teaching how to write good quality code by using "contracts" (pre/post-requisites, invariants, assertions). Students are invited to participate to the pilot by assigning 30% of the final grade to the homeworks. Yet, not all the students are participating. This is probably due to the fact that this group of students were not asked to complete programming homeworks during the the first term.

	Programming 2		Programming Lab.	
	HW1	HW2	HW1	HW2
Functions tested	4	3	10	11
Tests developed	15	37	37	97
HW Submitted	45	36	24	24

We have tested two homeworks for each course and we are now just writing tests for the third (and final) assignment of both courses.

In this initial phase we have implemented the unit-tests machinery and we have solved the following problems:

segmentation faults whenever an invalid pointer (null or uninitialized) is dereferenced a segmentation fault happens. We track these errors by registering an interrupt handler that jumps to a recovery routine when a SIG_SEGF interrupt fires. As the memory space of the program could be badly ruined, we run each test in a separate process and wait for its termination. If a SIG_SEGF happened the process is killed and the main program is signaled.

endless loops Each test is run against a timeout to catch endless loops.

exit and abort To avoid the termination of the whole tests we replace the 'exit' end 'abort' standard functions with a null implementation.

assert To track for correct usage of assertions we replace the standard 'assert' macro with a version that throws an exception that is expected by the tests. We can do that if the student's program can be compiled with the C++ compiler.

We are now compiling student's programs with a standard C compiler, and thus we cannot use exceptions. The 'assert' macro in this case sets a global variable and the tests check for it after the test.

C++ vs. C Students are asked to implement their functions in standard C. To allow us the usage of AspectC and of CppUnit exceptions we are trying to compile their code with a C++ compiler (the Gnu compiler g++) . Several standard C constructs raise errors when compiled with g++, e.g. the implicit casts of malloc-ed data, but they can be solved by using the option **-fpermissive** that transforms most of these errors into simple warnings.

missing functions and typos Sometimes the student implements only part of the homework or writes a typo in the function name or prototype. In this case the test system would'nt pass the compilation because of the missing functions. We solve this issue by giving a dummy implementation of all the required functions that just throws an exception and writes a message ("Function XXX is not implemented.")

This dummy library is linked against the student code and test system with the **-kmuldefs** option that allows for multiple definitions of a function and links the first found.

We are just now introducing and running the aspect-based machinery to replace failed functions with their reference implementation. The machinery will be used on the third (last) assignment. All the tests run so far will be re-run to purge errors coming from functional dependencies to failing functions.

8. MORE IDEAS

Automatic generation of aspects

The aspect example shown in figure 3 suggests that the aspect sub-classes coud be automatically generated by using a standard naming scheme for function, tests and reference implementations (e.g. 'square', 'test_square', 'square_OK') and by extracting the functional dependencies from the student's code.

Checking the complexity of the algorithms implemented

Aspects are the perfect tool to add counters to functions. With counters we can track allocation and deallocation of dynamic data structures, or count how many times a function is called or a field is accessed. Thus we can compare the algorithms used by the students to see which one is more efficient than others', both respect to its complexity and respect to memory usage.

Reusing student's submissions

Student's solutions and tests are a valuable resource that should be exploited to enhance their learning opportunities. To this aim the teacher could enlarge the set of tests by selecting tests from the student's submissions.

Playing with the students

We would like to involve the students in the correction by running a tournament and by ranking their solutions and tests to highlight both the student's solutions that pass more tests and the student's tests that catch more errors. To have more fun the teacher's participates to the tournament with his reference solution and tests.

References

[1] Derek S. Morris. "Automatically Grading Java Programming Assignments via Reflection, Inheritance, and Regular Expressions", Proc. Frontiers in Education 2002, Boston, USA, 2002.

[2] `http://www.aspectc.org`

[3] `http://cppunit.sourceforge.net`

```
#include <stdio.h>
#include "main.h"

// Abstract aspect that tests a function and replaces it with a reference implementation if the test fails
aspect Redirect {
        // abstract pointcut that defines which function will be instrumented
        pointcut virtual           calls() = 0;
        // abstract method that calls the reference implementation
        virtual int                use_correct(int X) = 0;
        // abstract method that tests the function
        virtual bool                test_it() = 0;
        // abstract method that returns the function's name
        virtual char *         name() = 0;

        // variable to store the test outcome
        bool passed;

        // just before running the code, run the tests in the order given by the 'ordering' directives (see later)
        advice execution("% main(...)") : before()  {
                printf("testing the '%s' function ... ", name());
                passed = test_it();
                printf(passed ? "PASSED!\n" : "FAILED!\n");
        }

        // replace all the calls except when used in the unit tests
        advice calls() && !within("% test_%(...)") && args(X) : around(int X)  {
            if (passed) {
                 printf("\n\tusing ORIGINAL version of %s\n", JoinPoint::signature());
                 // proceed with normal execution
                tjp->proceed();
            } else {
                 printf("\n\tusing REPLACED version of %s\n", JoinPoint::signature());
                // replace the function with its reference implementation
                *(tjp->result()) = use_correct(X);
                }
        }
};

// we extend the abstract aspect Redirect to test the 'f' function
aspect Redirect_f : public Redirect {
        // as the 'f' function uses the 'square' one,
        // we define the proper ordering of tests, that is: first 'square' then 'f'
        advice main() : order("Redirect_square","Redirect_f");

        pointcut calls()                = call("% f(...)");
        char * name()                { return "f";                }
        int use_correct(int X)        { return f_OK(X);                }
        bool test_it()                { return test_f();        }
};

// we extend the abstract aspect Redirect to test the 'square' function
aspect Redirect_square : public Redirect {
        pointcut calls()                = call("% square(...)");
        char * name()               { return "square";        }
        int use_correct(int X)        { return square_OK(X);        }
        bool test_it()                { return test_square();        }
};
```

Figure 3: Aspect example.