Introducing Students to Professional Software Construction: A "Software Construction and Maintenance" Course and its Maintenance Corpus

Guy Tremblay, Bruno Malenfant, Aziz Salah and Pablo Zentilli
Dept. d'informatique, UQAM
C.P. 8888, Succ. Centre-Ville
Montreal, QC, Canada, H3C 3P8
{tremblay.guy,malenfant.bruno,salah.aziz}@uqam.ca

ABSTRACT

It is widely accepted that there is more to software construction than basic programming skills [13, 11, 15, 16]. Professional software construction involves not only understanding some theoretical concepts, but also mastering appropriate tools and practices. In this paper, we present an undergraduate course in *Software Construction and Maintenance*, developed with the goal of introducing students to those key concepts, tools and practices.

We first outline the content of that course, explaining how it fits within our undergraduate program. We then present a key element of that course—namely, its *maintenance corpus* along with its testing frameworks—used to concretely *introduce* students to various tools and practices, e.g., automatic test execution, build and configuration management, source code documentation, use of assertions, etc.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education

General Terms: Design

Keywords: Software Engineering, Testing, Maintenance

1. INTRODUCTION

Developing software in a professional manner requires not only knowing programming languages, but also mastering appropriate *practices and tools* [13, 11, 15], including those related with testing [11, 4, 16]. However, software construction related topics appear to be somewhat neglected in general software engineering books [18, 20], a fact which can also be noticed in the *Guide to the SWEBOK* related chapter's references [16].

For numerous years, *official* software engineering has been mostly concerned with documentation and process, largely emphasizing pre-construction phases (i.e., requirements analysis, specification and design), which has been reflected in most software engineering curriculum. For instance, our undergraduate program at UQAM is titled "*Informatique et génie logiciel*" (Computer science and software engineering).[1] As early as 1991, students had to take four (4) mandatory SE courses: *Requirements analysis and modeling*, *Design*, *Project management*, and *Requirements analysis and modeling project*. Since 1999, students must also take a *Formal specification* course [25]. Of course, students also take various CS/SE-related courses, e.g., discrete mathematics and statistics, data structures, operating systems, networks and distributed applications, database design and implementation, information systems, ethics and professionalism, etc. But except for the two introductory programming courses, there was, until recently, no course addressing software construction *per se* [16].

Over the last few years, with the emergence of the agile movement [3, 7], it is becoming accepted that using discipline in SE also means being concerned with good software construction practices. In the fall 2004, when revisions were made to UQAM's SE program, we decided to introduce a new course where students would be introduced early to some of the key practices associated with a disciplined approach to software construction [13, 11, 15]. In this paper, we present the *Software Construction and Maintenance* course that we developed for this purpose.

Outline of paper. Section 2 outlines the content of our *Software Construction and Maintenance* course. Section 3 presents a key element of this course, namely, the *maintenance corpus* used to *concretely* introduce students to various software construction practices and tools. Section 4 presents the two testing frameworks used within this maintenance corpus. Finally, Section 5 discusses how this course compares to other courses and how it relates to SEEK knowledge units [22].

[1]Before 2002, it was titled "*Informatique de gestion*" (Computer science and information systems), but its content was the same, only the name changed.

2. A "SOFTWARE CONSTRUCTION AND MAINTENANCE" COURSE

2.1 Course Outline

We designed our *Software Construction and Maintenance* course—taken during the third semester of a seven (resp. ten) semester regular (resp. Co-op) program—to act as a *bridge* between the basic programming knowledge developed in the (two) introductory programming courses and the abstract notions introduced in more advanced software engineering courses—more precisely, the design and project management courses. Our goal was to use a *bottom-up* approach where, for example, testing techniques and tools would be introduced and used by students in a first course, and then later discussed more *abstractly* (e.g., high-level test planning) in subsequent software SE courses.

Because the first two programming courses as well as the subsequent data structures course use object-oriented languages (Java and `C++`), we choose to use `C`, so students would be exposed to the imperative and procedural paradigm. Choosing `C` had the additional benefit that when students take the subsequent operating systems course, also in `C`, they know enough about the language to delve more deeply into programming with (Posix) threads [8], which was not possible before.

Here is the outline of the course:

1. Introduction to the `C` programming language.
2. Introduction to the use of Linux/Unix (basic shell commands, `gcc`, `make`, shell scripts [14]).
3. Basic software design concepts: Types of modules and components; abstraction (procedural, data, control); cohesion and coupling; encapsulation and information hiding; filters and pipelines.
4. Programming style, error handling, documentation tools (`DOC++`).
5. Debugging strategies and tools (`gdb` [14]).
6. Defensive programming, role and use of assertions (`assert` macro).
7. Tests: Black/white box tests, unit (module) vs. system-level tests. Test coverage and structural complexity. Testing scripts and frameworks (see Section 4).
8. Strategies, techniques and tools for performance evaluation and optimization (`gprof` [14]).
9. Configuration management (`CVS` [23]).
10. Software evolution and maintenance—of course, with the emphasis on *code maintenance*.

Few lecture hours are dedicated to maintenance. Instead, as described below, students are introduced to maintenance in a practical and concrete way.

2.2 The Place of Maintenance

Only *discussing* maintenance and related concepts would have been *insufficient*. Rather, we wanted students to understand what maintenance is really about... by *practicing* maintenance. Thus, the last practical assignment is a *maintenance assignment*—examples appear in Section 3.3.

Having students perform maintenance work requires having some software to maintain, complex enough for the maintenance task not to be trivial, yet simple enough that second year students can perform the required task. Thus, we decided to develop our own *maintenance corpus* (Section 3).

Having our own maintenance corpus makes it possible to introduce *concretely* key practices and tools, not only by teaching about those notions during lectures, but also by having students *use* such tools and practices. In other words, students must also learn about those topics by *reading* and *understanding* software (documentation and source code) developed using such practices and tools.

3. THE MAINTENANCE CORPUS

Our maintenance corpus needs to be at the appropriate level, i.e., neither too simple nor too complex (e.g., some real open-source software). To attain the correct level of complexity, we started... from some software developed by a group of four undergraduate students as part of a course project.[2]

3.1 Software for Managing a Personal Library

Our maintenance corpus software's goal is to manage the personal library of the first author, who wanted to keep track of the various books he was lending to colleagues and students. The key functionalities are the following, described using command-line calls—all identifiers have been *manually translated* in English, as our software is written in French:

- Borrow a book:

```
% books borrow Zentilli zentilli_pablo@yahoo.fr
        "Code complete" McConnell
```

- Return a book:

```
% books return "Code complete"
```

- Identify all books borrowed by some person:

```
% books borrowed Zentilli
```

- Identify the person who borrowed some book, then send an email asking that the book be returned:

```
% books borrower "Code complete"
% books recall-book "Code complete"
```

- Send an email to each and every borrower, asking that the borrowed books be returned:

```
% books recall-all-books
```

3.2 Improving the Initial Software

The initial software written by the four students (in `C`) was, indeed, *usable*. However, it was badly written, unstructured, with no comments, a lot of (cut-and-pasted) duplicated code, no tests, some unimplemented operations, etc.

Asking our students to modify directly that software would have made their job quite difficult. Furthermore, it would not have shown them any of the good practices associated with professional software construction, since none had been used... For instance, in the initial version, absolutely no tests had been defined. Since we wanted students to modify

[2]This course is not part of our bachelor program. Instead, it is part of a one year "CS Certificate," so students in that course have somewhat limited programming knowledge and skills.

and improve the existing program and always ensure it behave correctly, our first step was thus to develop system-level regression tests. Once this was done, additional modifications and improvements were made, not to make the program *perfect*, but rather to show, in various places, the correct use of key software design, construction and test practices. Overall, the following modifications were performed before the first "public" release:

- Support was provided for the automatic execution of system-level tests (for regression testing purpose, as described in Section 4.1).
- The source code was put under revision control, using `CVS` [23].
- The source code was minimally restructured (decomposed into a few modules) and an appropriate `makefile` was defined to automate the build process.
- Most header files were documented with `DOC++`[3]—`DOC++` uses special comments, *à la* `JavaDoc`, to document module interfaces, including (informal) *pre/post-conditions*.
- Assertions (mostly for pre-conditions) were added in some modules.
- Unit tests (using `MiniCUnit`: see Section 4.2) were (partially) defined for some modules.

3.3 Programming Assignments Using the Maintenance Corpus

The maintenance corpus is used in the final assignment. Students are provided with a document explaining the code improvement process described above, the resulting software structure, as well as the various tools not formally discussed in lectures[4]. On-line documentation (`DOC++`) is also provided, along with source code.

So far, the maintenance *corpus* has been used three times:

- Winter 2006: Students had to add a new `find` operation. More precisely, a preliminary version of that command was already available, but was implemented as a shell script, as it had not been included in the initial software release. The goal was thus to implement it like the other commands, in `C`, which required:
  - Modifying some system-level tests.
  - Writing new `C` code and modifying the `main` program.
  - Developing some unit tests.
  - Documenting the modified header files with `DOC++`.
- Summer 2006: Students had to improve the existing code, without changing its overall behavior, which required:
  - Removing *magic constants* and introducing appropriate symbolic constants.
  - Removing useless initializations.
  - Localizing the variables—variables were always declared at the top-level of functions.
  - Simplifying the error messages generation by defining additional error message routines.
  - Modifying the `main` program to use a *table-driven dispatcher* instead of a long, linear, `if` instruction.
- Fall 2006: Students had to modify the `recall-all-books` command so that the email sent to borrowers explicitly indicate the borrowed books list—initially, the email simply indicated that *some books* had been borrowed, without saying which. This required modifying the system-level tests as well as modifying some existing code. Other modifications also had to be made:
  - Reimplementing one routine unit tests, so that those tests use the `MiniCUnit` test framework instead of simple `asserts`—failure of the latter simply aborts program execution, thus producing an incomplete execution report (see Section 4.2).
  - Adding (`MiniCUnit`) unit tests for a routine which had no such test.

Other planned assignments are the following:

- Similar to the first assignment mentioned above, reimplement an existing command (`list`) which will require modifying existing system-level tests, adding new ones, and modifying/adding code.
- Add a new `return-books` command allowing to specify multiple returned books.
- Add a new `recall-books` command allowing, with appropriate options, either to require from a borrower that multiple books be returned or that all borrowed books be returned.
- Allow various commands taking title arguments to accept incomplete or partially specified string, for example, using regular expressions—in the current version, all such titles must match exactly.

A recurring theme in all those assignments is the need for testing, system-level testing as well as unit testing, a topic we now discuss.

4. THE TEST FRAMEWORKS

Tests can be performed at various *levels* [21, 6], for instance, unit tests (for independent modules and components), integration tests (for combination of modules and subsystems), system tests (for the whole system).

Various *test frameworks* have been developed, `JUnit`, popularized by proponents of *eXtreme Programming* [3], being the most well-known [5]. Such frameworks are now being used in various undergraduate courses [2, 10, 17].

Test execution frameworks are characterized, among other things, by the use of *assertions* to describe tests, by the fact that results are emitted (mostly) only when errors are detected, and by the support they provide to the (structured) organization of tests—for example, a hierarchy consisting of test methods, test cases, test suites.

For our corpus maintenance, we choose to develop our own test frameworks.

[3]`http://docpp.sourceforge.net`
[4]`http://www.info2.uqam.ca/~tremblay/INF3135/Biblio`

4.1 System-Level Testing of C Programs

The book management software described in Section 3 must manage persistent data. To make it simpler for students, persistent data is stored in *textual* databases. Performing system-level tests thus requires being able to compare both the results emitted on `stdout` and the textual modifications made to the database.

For this purpose, we wrote a `C`-shell script that works as follows. First, the script requires that *each test case* be specified using four (4) different text files—a complete test suite is composed of many different test cases:

- `test-X.commands`: A series of commands to be executed for this specific test case.
- `test-X.before`: The state of the database *before* execution of the test case.
- `test-X.results`: The output results expected from execution of the test case commands.
- `test-X.after`: The expected state of the database *after* the test case commands have been executed.

For each test case, for example `X`, the following operations are then performed :

- Define the current state of the database to be as described by `test-X.before`.
- Execute the commands in `test-X.commands` and save the results emitted on standard output in a temporary file `test-X.obtained`.
- Using `diff`, compare `test-X.results` and `test-X.obtained` and signal *significant* differences—by default, comparisons are performed leniently, i.e., differences pertaining to white spaces or letter case are ignored, although other comparison modes can be specified.
- Still using `diff`, compare `test-X.after` with the current content of the database and signal differences.

The final global output of the test script then indicates the total number of test cases that were executed as well as the number of test cases where differences were noted and, if any, whether differences were noted for the emitted results or database updates.

This testing approach can be generalized to other types of data. For instance, we recently added a log file that records the various recall emails sent to borrowers. Thus, two additional text files are now required to describe a test case: one describes the content of the log file before execution of the test case commands, the other describes its content afterwards.

Interestingly, the various `C`-shell scripts used for automating execution of system-level tests are also useful examples of shell scripts, a topic also covered in the course (use of Unix/Linux).

4.2 Unit Testing of C Modules

Our unit test framework, called `MiniCUnit`, was inspired by `MinUnit`[5], which defines a few *macros* and uses various variables to keep track of the number of tests executed, assertions evaluated, etc. `MinUnit` is quite minimal and, also, *fragile*, since test cases execution aborts as soon as an erroneous assertion is encountered. `MiniCUnit`, on the other hand, is more robust: at compile-time, through the use of `C` functions and a more limited use of macros, and at execution-time, since test cases execution can proceed even when a failed assertion is encountered.

[5]`http://www.jera.com/techinfo/jtns/jtn002.html`

```
#include "MiniCUnit.h"

static TestSuite suite1();

int main(int argc, char *argv[])
{
  executeTestSuite(suite1);

  printf( "%s", testsSummary() );
  return 0;
}

// Various test cases definitions.
TEST_CASE( test1 )
  ...
END_TEST_CASE
   .
   .
   .
TEST_CASE( testK )
  ...
END_TEST_CASE


// Test suite definition.
TEST_SUITE( suite1 )
  addTestCase( test1 );
  ...
  addTestCase( testK );
END_TEST_SUITE
```

Figure 1: Typical structure of a test program using `MiniCUnit`.

Figure 1 outlines a typical test program using `MiniCUnit`. The final execution summary, obtained from `testsSummary()`, has the following form:

```
*** Summary: 1 suites executed;
             6 tests executed, 0 failed tests;
             13 assertions evaluated, 0 failed assertions
```

Various forms of assertions are provided—e.g., `assertTrue`, `assertFalse`, `assertEqualStrings`, `assertEqualInts`. Each requires a string argument, used in case of failure. For example, suppose the following `assertEqualStrings` is the 5th assertion in test case `test2` within test suite `suite1`:

```
  assertEqualStrings( "dog", "cat", "Bad mix" );
```

Result for `suite1` execution would then include the following:

```
1) "suite1"::"test2"
    #5: 'Bad mix ("dog", "cat")'
```

5. CONCLUSION

In this paper, we presented a new *Software Construction and Maintenance* course which aims at introducing students to some of the key concepts, practices and tools of professional software construction. We also presented a key component of this course, namely, its *maintenance corpus* along with its associated test frameworks.

As we showed in Section 3, this corpus, on which students must perform concrete maintenance tasks, introduces students to various good practices and to the proper use of software construction tools, e.g., source code revision control (`CVS`), automatic build management (`make`) and execution of (system-level as well as unit) tests, semi-formal documentation of module interfaces (`DOC++`), use of assertions (`assert`).

Other courses discussing software construction and maintenance have been proposed. For instance, Postema et al. [19], as we do, discuss the practical aspect that should be present in a course addressing maintenance, whereas Austin and Samadzadeh [1] put more emphasis on the topic of software *comprehension*. Various courses related with testing have also been proposed [9, 12, 24].

As for "software construction" courses, a recent search on the Web[6] showed that *most* courses supposedly dealing with software construction fall into one of two categories:

- General SE overview courses, i.e., courses discussing software *development* in "general" (life cycle, requirements and design modeling, project management, etc.).
- Programming courses, i.e., courses dealing almost exclusively with (OO) programming-related concepts.

Only a small number of courses appear to have an orientation similar to our course, that is, attempt to address the various aspects of software construction (style, design, contracts, tests, configuration, tools) and maintenance, all within a single (early) course—a similarity which can also be noticed through the key references [13, 11, 15] provided in those courses' outlines.

More precisely, our course addresses the following SEEK Knowledge Units [22], thus showing its *breadth* when compared with "programming" courses:

- CMP.ct Construction technologies (1, 2, 5, 6, 7, 16)
- CMP.tl Construction tools (1, 3, 4, 5)
- PRF.com Communication skills (1)
- MAA.md Modeling foundations (2)
- DES.con Design concepts (1, 4)
- DES.str Design strategies (1)
- DES.ste Design support tools and evaluation (2)
- VAV.tst Testing (1, 2, 4, 5, 8, 10, 11)
- EVO.pro Evolution processes (1)
- EVO.ac Evolution activities (1, 2, 6)
- MGT.cm Software configuration management (1, 3, 4)

So far, feedback from students on this *Software Construction and Maintenance* course has been very positive: in a recent (oral) survey where over 100 students were interviewed, it scored among the top third best-evaluated courses. More precisely, on a 5 points scale—where 5 is the best score and 1 is the worst—, the course obtained an average score of 4.39 for its "quality" and an average score of 4.24 for its "usefulness"—how useful the students think what they learned in the course will be.

As future work, we plan to extend our maintenance corpus to incorporate additional tools:

- `C`-style checking tool, to identify potential places where the source code style could be improved.
- Source code static analysis tools, e.g., data flow analysis as performed by `Uno`[7], to improve the *semantic* quality of the code.
- Test coverage tool, e.g., `gcov`, to measure and improve the quality of tests, both system-level and unit tests.

Our hope is that, after taking this *Software Construction and Maintenance* course, students will come to appreciate how the use of tools can help them perform their job in a more professional manner, appreciating also the role testing tools can play in developing and maintaining software.

[6]Note that we did not find any *paper* discussing explicitly software construction, as it is now understood [16].

[7]`http://www.spinroot.com/uno`

6. ACKNOWLEDGMENTS

Thanks to E. Chieze, I. Maffezzini and the anonymous referees for their feedback and suggestions.

7. REFERENCES

[1] M. A. Austin and M. H. Samadzadeh. Software comprehension/maintenance: an introductory course. In *ICSEng 2005*, pages 414–419, 2005.

[2] E. Barriocanal, M.-A. Urbán, I. Cuevas, and P. Pérez. An experience in integrating automated unit testing practice in an introductory programming course. *SIGCSE Bulletin*, 34(4):125–128, 2002.

[3] K. Beck. *Extreme Programming Explained—Embrace Change*. Addison-Wesley, 2000.

[4] K. Beck. *Test-Driven Development—By Example*. Addison-Wesley, 2003.

[5] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[6] A. Bertolino and E. Marchetti (Ass. Eds.). Software testing. In *Guide to the SWEBOK (2004 Vers.)*, pages 5.1–5.16. IEEE Comp. Soc. Press, 2004.

[7] B. Boehm and R. Turner. *Balancing Agility and Discipline—A Guide for the Perplexed*. Addison-Wesley, 2004.

[8] D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[9] H. B. Christensen. Systematic testing should not be a topic in the computer science curriculum! *SIGCSE Bulletin*, 35(3):7–10, 2003.

[10] M. Goldwasser. A gimmick to integrate software testing throughout the curriculum. *SIGCSE Bulletin*, 34(1):271–275, 2002.

[11] A. Hunt and D. Thomas. *The Pragmatic Programmer—From journeyman to master*. Addison-Wesley, 2000.

[12] F. Kazemian and T. Howles. A software testing course for computer science majors. *SIGCSE Bull.*, 37(4):50–53, 2005.

[13] B. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[14] M. Loukides and A. Oram. *Programming with GNU Software*. O'Reilly, 1997.

[15] S. McConnell. *Code Complete—A Practical Handbook of Soft. Constr. (Second Ed.)*. Microsoft Press, 2004.

[16] S. McConnell, T. Bollinger, P. Gabrini, and L. Martin (Ass. Eds.). Software construction. In *Guide to the SWEBOK (2004 Vers.)*, pages 4.1–4.10. IEEE Comp. Soc. Press, 2004.

[17] R. Noonan and R. Prosl. Unit testing frameworks. *SIGCSE Bulletin*, 34(2):232–236, 2002.

[18] S. Pfleeger. *Software Engineering—A Rigorous and Practical Approach (Second Ed.)*. International Thomson Computer Press, 2001.

[19] M. Postema, J. Miller, and M. Dick. Including practical software evolution in soft. eng. education. In *14th Conf. on Soft. Eng. Educ. and Tr.*, pages 127–135, 2001.

[20] R. Pressman. *Software Engineering—A Practitioner's Approach (Fifth Ed.)*. McGraw-Hill, Inc., 2001.

[21] P. Robillard and P. Kruchten. *Software Engineering Process with the UPEDU*. Addison-Wesley, 2003.

[22] The Joint Task Force on Computing Curricula. Software Engineering 2004—Curriculum guidelines for undergraduate degree progams in soft. eng. Aug. 2004.

[23] D. Thomas and A. Hunt. *Pragmatic Version Control Using CVS*. The Pragmatic Bookshelf, 2004.

[24] A. Tinkham and C. Kaner. Experience teaching a course in programmer testing. In *Agile Conf. 2005*, pages 298–305, 2005.

[25] G. Tremblay. An undergraduate course in formal methods: "Description is our business". *SIGCSE Bulletin*, 30(1):166–170, 1998.