Introducing Testing Practices into Objects and Design Course

Ellen Francine Barbosa
José Carlos Maldonado
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
{francine, jcmaldon}@icmc.sc.usp.br

Richard LeBlanc
Mark Guzdial
College of Computing
Georgia Institute of Technology
{rich, guzdial}@cc.gatech.edu

Abstract

Though software testing courses are commonly taught as part of Software Engineering curricula, software testing is still a challenging issue in Software Engineering education. Students frequently see testing only as something that happens at the end of the development process. Two challenges can be recognized: “How to make the students recognize the relevance of the testing activity?” and “How to motivate the students on using testing ideas in their projects?”. In an attempt to explore the impact of introducing testing practices throughout development, during the past Fall semester we modified the project requirements in a course on object-oriented analysis and design offered to the undergraduate students in Computer Science at Georgia Institute of Technology. Our idea was to require the students to start thinking about testing as early as possible, by including testing-related practices in all phases of the development process. This paper presents the details of the testing approach used in the course and discusses the results we obtained, in terms of the students’ attitudes and learning.

Keywords: Software Testing Education, Equivalence Partitioning Testing Criterion, Functional Testing, Object-Oriented Development Process.

1: Introduction

Software testing is one of the most important activities to guarantee the quality and the reliability of the software under development [3, 4, 10, 12, 14]. Thus, the development of students’ testing skills should play a fundamental role in meeting the increasing demand for qualified software engineering professionals. However, despite the efforts of the academy to motivate the students regarding its relevance, software testing is not a popular discipline and, in fact, many students seem to believe that testing is only something they have to do to show that their programming assignments work as required.

Traditionally, the testing has been taught according to the classical approach to software development, the waterfall model, where the phases for constructing the system are described in a linear way (from analysis and requirements gathering, to design, to coding, to testing, and to code release) and the tests just take place at the end of the process. Also, the more pragmatic issues of testing are often addressed only in advanced and specific courses on Software Engineering, for seniors or graduate students.

A different strategy for teaching software testing would be to introduce some testing practices as early as possible, during all the phases of the development process, based on a more practical
perspective. Modern experience has already suggested that consideration of testing throughout the development phases provides a way to enhance a system’s quality and to reduce development costs. The most prominent current example of this thinking is eXtreme Programming (XP) methodology [1] (http://www.xprogramming.com) – a relatively new development approach which proposes, among other core practices, to write tests even before writing code (“test first, code later”). Applying this idea in the educational setting can be seen as an interesting way to create new learning opportunities. Besides that, introducing ideas from modern methodologies (like XP) may serve to provide extra motivation to the students.

The work described in this paper was a first attempt at introducing some basic testing practices into an undergraduate course at Georgia Tech on object-oriented design – CS2340: Objects and design. The main topic of the course is an object-oriented development process in which students are required to practice the fundamentals of analysis, design and programming by means of a design project assignment, involving a real world situation. During the most recent offering of this course, students were required to incorporate some testing practices from the beginning until the end of their projects. The idea was to provide the students with a more pragmatic view of testing as well as to encourage them to think about testing earlier, during all phases of the development process.

The remainder of the paper is organized as follows. In Section 2, the main aspects of the CS2340 course are discussed. Section 3 describes some ideas to fit testing practices into the scope of the course. Section 4 evaluates the effects of our testing approach on the students’ learning. Finally, in Section 5, our conclusions and further plans are presented.

2: CS2340: Objects and design

The main goal of the CS2340 course is to explore higher-level issues of analysis and design, with some emphasis on user interface development. It is a one semester course and is offered to the second-year undergraduate students on Computer Science, at Georgia Institute of Technology. Projects are team-oriented, involving the development of a small-to-medium sized object-oriented system, built upon a real world problem. The past Fall semester, for instance, the topic for the course was Genealogy Information (http://coweb.cc.gatech.edu/cs2340). The 160 students had to create a tool that collects genealogical information, notes inconsistencies and flaws in the database, provides some graphical representations, and supports a standard genealogical information format.

As pre-requisites, students taking the CS2340 course have already had:

- one semester on Introduction to Computing, focusing on the design, construction, and analysis of algorithms;
- one semester on Object-Oriented Programming, in Java; and
- one semester on Languages and Translation in C, using tools like LEX and YACC to explore the issues of language implementation, from models of the bare processor up through tokenizing and parsing.

Before or concurrently with the CS2340 classes, students are also required to take one semester Software Practicum, where the basics of Software Engineering, including a general view on software testing, are presented.

The design process taught in the CS2340 course encompasses the traditional phases of the object-oriented development, starting from analysis and leading through design and implementation. The process is not linear; students are expected to go back and forth between analysis and design, design and programming, and even analysis and programming phases.
In the analysis phase, students are required to elaborate scenarios and CRC (Class-Responsibility-Collaborator) cards [2]. From a development perspective, “playing” with CRC cards represents a useful way for early capturing the design decisions. They help to define the responsibilities for a given class as well as the interactions between the classes while performing several scenario tasks. Actually, they correspond to an informal approach to object-oriented modeling. From an educational perspective, CRC cards can be easily shared in groups, what is especially useful for the CS2340 team-projects. Besides that, since they are non-technical, there is no language or notation to be learned.

In the design phase, students have to develop UML class diagrams [5] for the system, defining the attributes and services of each class and formally identifying the connections between them. A detailed description of what each service is supposed to do is also required.

In the programming phase, students experience a variant of Smalltalk-80 – the Squeak language – in order to implement the system. In general terms, Squeak (http://www.squeak.org) [7, 11] is an open source and highly portable language, providing a rich multimedia support. By offering a good infrastructure for interesting and complex projects, Squeak is particularly appropriate for teaching Computer Science [8]. In the context of CS2340 classes, it was first used in 1998 and, since then, the experience has suggested that students can complete in a single semester more sophisticated and interesting projects in Squeak than in other languages, such as Java and C++ [8].

Regarding testing activity, despite its relevance in the development process, no systematic method to test the systems was adopted in the previous terms of the CS2340 course. Actually, the past Fall semester was the first time we explored testing practices into the scope of this class. Our testing approach and the results we obtained are described in the next sections.

As a final remark, it is important to notice that the CS2340 course has traditionally been served as a “laboratory” for applying and evaluating new approaches to teaching and learning in Computer Science. Two successful examples on that are the use of CoWeb [6] to promote joint learning; and the introduction of Squeak as a new way to teach user interface design and implementation. In this sense, our attempt to explore a different strategy for teaching software testing follows the established tradition of the course.

3: Testing approach for CS2340

In an attempt to fit testing practices into the scope of the CS2340 course, we chose to work with the Equivalence Partitioning criterion, built upon the functional testing idea. The purpose of functional testing is to find discrepancies between the actual behavior of the implemented system’s functions and the desired behavior as described in the system’s functional specification [13]. In this sense, there is a general agreement on Equivalence Partitioning as a key criterion for functional testing. Also, the “partitioning idea” can be easily understood even by students who have had little experience with testing. For these reasons, Equivalence Partitioning seemed to be a good starting point for use of testing in the course.

Basically, the criterion consists on dividing the space of possible program inputs into a finite set of partitions, and then selecting test cases from each equivalence class of the partition. An equivalence class defines a set of data values – valid and invalid ones – for which the program will perform the same computation. Every member of an equivalence class is said to be equivalent, with respect to some input condition, to every other member. Accordingly to Ostrand and Balcer [13], the idea behind each equivalence class is that all its elements are essentially the same for the purposes of testing. If the testing’s main emphasis is to attempt to show the presence of errors, then the assump-
tion is that any element of a class will expose the error as well as any other one. If the testing’s main emphasis is to attempt to give confidence in the software correctness, then the assumption is that correct results for a single element in a class will provide confidence that all elements in the class would be processed correctly.

In the educational setting, the Equivalence Partitioning has generally been taught in terms of small programs, dealing with numerical input values. Using the criterion to test this kind of program is straightforward. However, when the input domain involves more complex types of elements, as in the case of genealogy information, its application may be not so clear. Next we present a small set of directions we established aimed at helping the CS2340 students to apply the Equivalence Partitioning in their projects. Notice we are not suggesting that these directions represent a new method of input space partitioning. Rather, we include them just to provide the details of our approach to guide the students throughout the “partitioning idea” application.

3.1: Directions for applying the equivalence partitioning criterion

Our directions are based on the main activities in the application of Equivalence Partitioning: (1) definition of input conditions; (2) identification of valid and invalid equivalence classes, and (3) definition of test cases to cover the equivalence classes.

Defining input conditions

The definition of input conditions takes place in the OOA phase, at the same time of the development of CRC cards. Based on the idea that input conditions are closely related to the input data provided by the user, we encourage the students to carefully analyze the set of responsibilities related to each CRC card, looking for “hints” on the possible input data items that each class may be handling. It is important to highlight that, although the CRC cards do not explicitly deal with the input data items, analyzing their responsibilities under a macroscopic perspective can provide some insight into that. Additionally, since each class can interact with others, the collaborators should also be investigated in order to find the right class where a specific input item is being treated.

After having identified the input data items, students are required to inspect them in terms of characteristics and interactions. In fact, each input item has specific characteristics related to it, which can directly affect the system’s behavior. Furthermore, input items can interact with each other, also resulting on different values in the output domain. Ultimately, the set of characteristics and interactions related to all input data items can be seen as the input conditions for the system.

Our directions for defining input conditions are the following:

1. Analyze each CRC card, looking for “suggestions” on the input data items provided by the user.
   
   (a) Consider the responsibilities in a macroscopic way. Focus on what each class really has to do.
   
   (b) Analyze the responsibilities of the collaborators too.
   
   (c) Write down all the input data items related to a specific class.

2. Identify the main characteristics of each input data item.

3. Identify the interactions among the input data items. Think about how a specific input entry can be related to the other input data of the system. Focus on the system’s operation (scenarios can help on this).
4. Write down the characteristics and interactions of all input data items related to all classes of
the system. They will correspond to the input conditions for the program.

It is worth noting that while CRC cards are helpful to define input conditions, it is also the case
that input conditions can be useful in checking the consistency of the CRC cards with respect to the
responsibilities and interactions among classes they are covering.

**Identifying equivalence classes**

The equivalence classes are identified in the OOA phase, after students having defined the input
conditions for the system. Actually, the valid and invalid equivalence classes are established by
analyzing the input conditions, in terms of the correct and incorrect values needed to cover them.
However, before analyzing each input condition separately, it is necessary to consider the entire set
of input conditions defined. If two or more conditions are closely related to each other, it can make
more sense to join them into a single condition. Conversely, if a condition is too broad, it can be
necessary to refine it by creating two or more specific ones.

The directions for identifying equivalence classes are the following:

1. Analyze the set of input conditions for the system.
   
   (a) Look for related conditions. Try to join them.
   
   (b) Look for broad conditions. Try to refine them.

2. Analyze each input condition separately.
   
   (a) Think about the valid values to satisfy the input condition. They will correspond to the
       elements of a valid equivalence class.
   
   (b) Think about other possible values associated to the input condition, i.e., the invalid
       values. They will correspond to the elements of a invalid equivalence class.

3. Enumerate the equivalence classes, assigning a unique number to each class.

It is also important to highlight that the definition of input conditions/equivalence classes in the
same context of CRC cards and scenarios, during the analysis phase, can provide to the students a
better understanding of the requirements for the system they are developing.

**Defining test cases for the equivalence classes**

The definition of test cases to cover the equivalence classes requires a more detailed knowledge
on the classes (and objects) in terms of structure, attributes, and services that need to be provided.
For this reason, it makes sense to define the test cases at the same time as the development of UML
class diagrams, in the OOD phase, when the students have to deal with more detailed issues of the
design (such as instance variables and methods for the classes).

Still in the OOD phase, students have to elaborate a simple test plan providing information on
the test suite they have developed. For instance, they should provide information on the purpose
of the test case, the system’s class it tests, the equivalence class it covers, the expected result for
its execution as well as the obtained result (which should be filled in after the test case execution).
Ultimately, test plans are useful mechanisms to the development of well-documented test suites,
serving as a basis for the regression tests.

The directions for defining the test cases to cover the equivalence classes are the following:
1. Derive test cases associated with valid classes to cover all of them. A test case can cover a
   number of valid equivalence classes, as large a set as possible.

2. Derive test cases associated with invalid classes to cover all of them. A test case should be
   written for each invalid equivalence class.

3. Write a test plan providing information on the test cases.

Since the Equivalence Partitioning application takes place along all phases of the design process,
any change in the scenarios, CRC cards or UML class diagrams requires the input conditions,
equivalence classes and test cases be reanalyzed. Test plans should also be revisited in order to
reflect the eventual changes.

Observe that at the end of the OOD phase a number of test cases have already been written, even
before the students have started coding. Actually, the idea is to get the students involved with testing
practices as early as possible in the development process. Writing test cases even before writing
code forces developers to think about the functionality they are designing. They have to clearly
state the context in which the functionality will run, the way it will interact, and the expected
results. Although we are not applying the XP methodology, in some sense we are exposing
the students to one of its core practices: “test first, code later”.

After defining the test cases, students have to execute them against the system under develop-
ment. Squeak’s environment allows to structure, describe the context of test cases and run them
automatically by means of a unit testing framework – SUnit [1]. In general terms, S-Unit is a min-
imal but powerful framework that supports the creation and execution of test cases and test suites.
S-Unit can also serve as a mechanism to motivate regression testing: students create a repository
of test cases for their classes and, over time, this can yield a growing suite of regression tests that
represents the expected functionality of the system at any given time [9].

As a final remark, although our approach focuses on the functional testing ideas, students have
also been motivated to keep defining test cases while coding the system, based on its implementa-
details. However, no specific structural testing criteria have been adopted. This point should be
investigated in the future offerings of the CS2340 course.

The testing approach discussed in this section was evaluated in terms of its effects on the stu-
dents’ learning. The results we obtained are analyzed next.

4: Evaluation

The effects of our approach on the students’ learning were evaluated by means of a specific
question on the final exam and, mainly, by applying a voluntary survey to the students after they
had finished their projects (thus, just before the final exam).

The final exam question required the students to construct a test plan for a matching algorithm
drawn from the genealogy domain they had worked with in their project. The results on this question
were somewhat disappointing. Only about a quarter of the answers earned a score above the average
for the whole exam. Most fell in a middle range about 10% below the overall exam average score.
The answers indicated that most students had a reasonable idea about what they were supposed to
be doing, but many didn’t follow through on the details very well.

The survey was composed by four questions, covering the students’ attitude toward: (1) the
XP idea of starting testing early; (2) using testing criteria; and (3) writing test plans. 105 students
answered the survey. On the survey question regarding to the XP testing ideas, we gave the students
some statements related to when to start thinking about testing during the development process, asking them to choose all the options they agreed with. 12 students indicated that they prefer to deal with testing after coding the system; 53 agreed on applying testing practices through all the phases of the development process; 41 considered that starting thinking about testing in the analysis phase was helpful to better understand the system’s functionality; and 23 felt it helpful to design a set of test cases at least before the implementation phase.

Clearly, the students had a positive attitude toward the idea of starting testing activities as early as possible, which suggests that further attempts should be made to integrate this approach into early courses as a way improve student attitudes toward testing and their ability to do it effectively.

On the question about testing criteria, we tried to explore their use related to some aspects of the development process: confidence about test cases; confidence about system’s quality; motivation to think about errors; and reduction of the total development effort. Not applying any specific testing criterion was also considered. 46 students answered that they would rather test the systems by their “own way”, without using a specific testing criterion; 21 felt more confident about the test cases when working with some testing criterion; 36 felt more confident about the system’s quality; 29 felt motivated to think about common errors that could be committed during the software development; and 10 agreed with the reduction on the total development effort by applying a testing criterion.

Refining our analysis, we observed that from the 46 students who had a negative attitude toward testing criteria, 35 were strictly “against” the idea of working with them, while 11 students agreed they could be useful in some sense, mainly in terms of thinking about common errors. Actually, from the total of 105 students, 70 agreed on the relevance of applying testing criteria for some aspect of the development process.

In terms of test plans, two questions were applied. In the first question, the students were asked to rate their opinion of having to do a test plan, using a scale from 1 to 5 (1 means “a completely worthless idea” and 5 means “a really valuable part of the development process”). In the second, we provided them with some statements on how in the development process the test plans could be valuable; and for which kind of systems the test plans would be more useful. They were asked to mark all of the statements with which they agreed. For 62 students, having to elaborate a test plan was a completely worthless or a worthless idea; 31 students presented a neutral position; and only 12 students considered test plans as a valuable or really valuable idea.

Interestingly, from the 62 students who had a negative attitude toward test plans, 40 of them had also a negative attitude toward testing criteria. Our hypothesis is that the negative results about test plans can be related to the application of the Equivalence Partitioning criterion, which is not so suitable for testing systems whose input values are not numerical. This suggests the need to improve our directions for applying the criterion.

It was also notable that 54 students, especially those who had negative (37) and neutral (15) attitudes, agreed with the usefulness of elaborating test plans when developing complex systems. In fact, another reason for the “high rejection” of creating test plans can be related to the kind of system (small-to-medium size) the students had to develop in the CS2340 projects.

Finally, 13 students agreed that test plans could be valuable as a way to keep track of test cases, and 16 pointed out that they could be valuable if changes in the system were required. Maybe, had the students had to work in a project where more changes in the system were required, we had obtained different results. Besides that, we noticed that some of these students were the same who had a negative attitude toward test plans. This suggests that although this specific group of students had been “against” the test plans idea, they were able to recognize its usefulness for some aspects of the development process.
5: Conclusion and further work

In this paper we discussed some ideas regarding the introduction of testing practices in a course on object-oriented analysis and design, required for undergraduate students in Computer Science at Georgia Institute of Technology.

A key point of our work was to encourage the students to use testing practices as early as possible, through all the phases of the development process. In some sense, our approach focused on the idea of a “test-driven development”, which is one of the main practices of the XP methodology.

The students’ attitudes toward the testing approach were evaluated mainly by means of a survey. In general, although the students have shown a negative attitude on using the Equivalence Partitioning and on having to elaborate test plans, they clearly approved the idea of starting testing early. From this results, we are motivated to continue investigating the approach in future offerings of the course. Especially, we intend to introduce S-Unit early in the development process, as well as to explore the use of other testing criteria in order to produce more significant results.

Acknowledgments

This research has been supported by grants from FAPESP, CNPq and CAPES (Brazilian funding agencies), and from the National Science Foundation (NSF) and the Mellon Foundation.

References