9th Workshop on Teaching Software Testing (WTST 2010)
January 29 – 31, 2010, Melbourne, Florida

Experience in Teaching Test-Driven Development Course

Nawwar Kabbani
Florida Institute of Technology
150, W. University Blvd.,
Melboure, FL 32901
nkabbani2008@my.fit.edu

Cem Kaner
Florida Institute of Technology
150, W. University Blvd.,
Melboure, FL 32901
kaner@kaner.com

Abstract
At Florida Tech, undergraduate and graduate students take the course "Software Testing 2" to learn test-first programming by exploring structural (glass-box) methods in software testing such as unit testing and structural coverage, as well as other agile development methodologies such as refactoring.

This paper presents an experience in teaching Software Testing 2 course at Florida Tech by describing the teaching methods and materials; discussing the challenges and problems we faced in the course; and showing the common student mistakes patterns. We hope by presenting this to open a constructive critical discussion about teaching this kind of course in order to improve the teaching methods and reach a better formula for this course that can be adopted as well in similar courses taught in other institutions.

I. Introduction

Course CSE 4415/SWE 5415 is offered for both undergraduate and graduate students at Florida Tech. It is an elective for students in the M.Sc. in Software Engineering program, and a required core course for the B.Sc.. Cem Kaner is the course instructor. I was a student in Fall 2008 and the Teaching Assistant for the Fall 2009 semester. This paper presents our experience in teaching the course in the Fall 2009 semester. The class in 2009 had 18 students including two graduate students. The background of 2009 students suggested that they were generally better than the average of previous years students.

The Course is described in the university catalog as:

> *"SOFTWARE TESTING 2. Explores structural (glass box) methods for testing software. Testing of variables in simultaneous and sequential combinations, application programmer interfaces, protocols, design by contract, coverage analysis, testability, diagnostics, asserts and other methods to expose errors, regression test frameworks, test-first programming."*

Next subsection lists the course objectives. Then we talks about the textbook used in the course and explain how and why it was picked. Section II presents the course contents of topics, labs, homework and assignments. Section III shows some of the common students mistakes in their work. Then section IV

discusses some of the resistance problems we faced during the course. Finally, we summarize the paper with some conclusions and recommendations about the course and future versions of it.

Course Objectives

The objectives of the course as defined in the syllabus are:

- Relearn basic programming skills through a lens of writing code that always works -- move from programming to software engineering.
- Adopt industry standard development tools and use them effectively,
- Improve your cognitive skills in basic development, especially problem decomposition and technical (programmer to programmer) communication
- Learn glass box test techniques
- Familiarity with the agile approach to development lifecycles, and gain some experience with agile approaches
- Work through exercises, and develop work products, that I think are likely to help you interview for new positions and do well in your first year on the job.

The first objective states that students will "relearn" programming skills. That is because much of the new development style is different from what they might have learned before in traditional programming courses. This requires a fair bit of discipline and motivation from the student. As we discuss later in this paper, students have shown some forms of resistance to the new techniques, such as writing tests after code not before, or little appreciation of a lot of code refactoring techniques.

To achieve the second course objective, students had to learn and use industry-level development tools. This includes: Eclipse for an Integrated Development Environment, JUnit for unit testing framework, Subversion for version control system. ECL-Emma[8] for structural coverage and Eclipse checkstyle for code style.

Textbook

Students come into this course with a variety of backgrounds. The students in Fall 2009 were mainly seniors at Florida Tech, most of whom were adequate or good programmers. The students in Fall 2008 were more mixed—several graduate students had much weaker skills and needed a textbook, a progression of simple-to-difficult exercises, and personal coaching to either develop skills that they apparently never developed in school or that they forgot as they spent years working in industry. Dr. Kaner chose to use an introductory programming book for the course because the more advanced books on unit testing with jUnit were too difficult for the students or out of date. We did review one jUnit book [1] at the start of the class and several students felt it was unworkably difficult for them.

Experience from last year in using Langr's *Agile Java* [3] as the course textbook was rather negative for several reasons.

- The book's first several chapters—including their exercises—were cumulative, making it difficult to change the order of topics.
- The exercises were long and tedious. Unfortunately, to be able to tackle an exercise in Chapter N+1, students needed their code from almost all of the exercises in Chapter N.

- The examples focused on management of data (storing data in databases; modeling the game of chess as a data problem). It took many chapters before students felt as though they were actually doing something.
- Basic time-saving concepts (especially, loops) were presented relatively late in the book, making tasks in earlier chapters tedious. Students felt as though they were seeing (or doing) tasks in a stupidly wasteful way and were then required to refactor code because it was so poorly done. This detracted from credibility and motivation for refactoring.
- The book emphasized the use of jUnit as a tool but taught almost nothing about unit test design.
- The book emphasized the use of refactoring but gave little guidance on how to refactor or how to spot issues that needed refactoring.

In addition, I felt that Langr's exercise designs were excessively prescriptive. It told the student in detail what class to use, how to name it, etc. This probably helps instructors set up automated grading of assignments, and it was probably necessary to Langr because he had students start their Chapter N+1 code from Chapter N—variation in implementation across students could make this impossible. But the effect on us as students was that it killed our creativity, forced us to follow design decisions that seemed poor (and were poor—we refactored the code later), and prevented us from thinking through anything but the mundane details of the implementation. I think this alienated several students in the course.

For 2009, we decided to find a different text. Our requirements (strong preferences) were:

- Introduce principles of OOP right in the beginning since unit testing in Java requires basic understanding of objects and classes.
- Include small projects with diverse kinds of problems that are suitable for learning different glass-box testing techniques.
- Introduce testing ideas early on, and not make the kinds of absurd suggestions (such as, "test everything") that are common to many introductory books.
- Ideally, give some practical suggestions on test design.
- Explain the use of a debugger, rather than just telling students to use one.
- Provide support for all of the learning objectives of the course.

We considered several books on Java, C#, and Python. The best match for our requirements was Barnes and Kolling's *Objects First with Java* [1]. This book was better than Langr, but it had its own problems:

- The book was tightly integrated with BlueJ as an IDE. One of the course objectives was to use more sophisticated tools (Eclipse and its plug-ins), so we had a mismatch between the book and the course content. We added a small amount of in-class coaching on using Eclipse. This was a minor problem for the 2009 students, but it would have been a more significant problem for several of the 2008 students—we expect the 2010 student pool to look much more like 2008 than 2009.
- The book introduced unit testing a little later than we would have liked. Also, it did a good job introducing debugging, but its testing lessons were concise and inadequate. We therefore addressed unit testing, refactoring, and other skills via lecture. For 2009, this was not a problem. Most students were able to learn the material from the lectures and slides. However, some of the

students didn't learn the material well this way, and many of the 2008 students seemed to need or want the more comfortable support that can be provided by a textbook.

- In the first chapters, the book's exercises were simple and self-contained enough to provide good starting points. We modified them in ways that were more effective for introducing test-driven development, but the book's descriptions of the original exercises provided a useful starting point for the TDD-modified exercises. The 2009 students had no trouble with this. We think this will work for 2008-level students as well. In later chapters, our priorities and the book's diverged to the point that we abandoned the book. This is not a criticism of the book. It's an introduction, and we were teaching more advanced material. But the effect was that after about a month, the security blanket of a textbook was gone. Students who needed that level of support lost it.
- Because we modified our exercises from the book or created new ones, the book provided students with no feedback about the quality of their homework. We reviewed every submission and made comments on all of them, either directly on the paper or by sending email to the class identifying commonly-made errors in Assignment X, or by discussing them in lecture. This required an enormous amount of grading time and it was not fully effective. Several students understood what they did wrong but didn't understand what to do in order to do it right. We rarely provided a step-by-step presentation of how we solved the assignment, but several students would have benefitted from this as a form of feedback.

II. Course Contents

This sections presents some of the main parts of the course contents. First we list the key topics that were part of the class lectures, then we talk about the labs and homework, then we discuss the final exam and show some interesting findings about it.

Key Topics

We emphasized the following topics:

1. Basics of unit testing.
2. Refactoring.
3. Support tools for software engineering (debugging, source control, coverage monitors, checkstyle).
4. Unit test design for different types of control structures.
5. Test design for data structures.
6. Test design for boolean expressions.
7. Structural coverage.
8. The design of random number generators and the use of RNGs to create test data.
9. Creation and management of files of test data and files of test oracle data.
10. System-level testing strategies that test architects would build tools to implement (such as high-volume test automation based on oracles, or based on diagnostic probes, or based on model-derived stored data).

Labs

The course included seven in-class labs. Some of the labs are meant to let the student do some brainstorm about new concepts they are about to learn. And by that they can reach some conclusions on their own,

and they can be more ready for reception and analysis of the new ideas. Labs ranged between 15 and 50 minutes long. And students had to work in groups of 2 to 4 persons per group.

1. The first lab students were asked to give and discuss testing ideas for basic code structures such as loops and conditionals.
2. In the second lab, students were asked to think about how to refactor their first homework to allow easy changing of the data structure from array to hash-table or vice versa. This was their first work with refactoring, and it was meant to let them understand its purpose.
3. Third lab was training on problem decomposition and iterative test-first programming. Students were asked to decompose a problem whose solution their familiar with. And that was Insertion Sort. They were asked to do it in test-first style and show iterative evolution of the solution.
4. Lab 4 was debugging. This was the only time they were explicitly asked to do debugging which is an important skill they were required to learn in the course. In this lab, students were provided with code that was poorly written and contained a lot of unintentionally inter-dependant unit tests.
5. Lab 5 was about testing common data structures (arrays, lists, maps, etc.) taking into account coverage metrics.
6. Labs 6 and 7 were about refactoring. As we show later in this paper, students seemed to misunderstand refactoring. These two labs were created to address this point. In these labs students were given code that contained smells, and they were asked to find them and suggest refactoring techniques to solve them.

Assignments and Homework

The course included two assignments worth 20% of the grade, and five homeworks worth 25% combined with the labs. An assignment is intended to be bigger than a homework which is intended to be weekly.

The appendix include the homework texts of the course. The first homework was a simple "Hello World!" program to let the write their first test-driven code and get familiar with JUnit and the development environment and tools. This homework went well. Some students had technical problem with the setup of the tools. However, the code submitted by most students met the homework objectives.

Second homework was designed to get experience with testing basic control structures such as conditionals and using code coverage tools alongside. In this homework, several students were still ignoring checkstyle warnings or not using it at all. Some also did not pay enough effort to reach high coverage rates.

The third homework, which was about Date Arithmetic, involved testing of more complex control structures, like loops and Boolean expressions. The most common problem students faced was choosing their oracles. That's because Java lacks built-in support for some date arithmetical operations (such as difference between two dates).

The fourth homework involved working with files and exceptions. The homework itself didn't require implementing complex logic. But it required a little bit of understanding of the purpose of testing. Unfortunately, most students chose a naïve way in testing. For example, as we show in section III, they didn't know how to pick reasonable oracle. Many also showed some weakness in dealing with exceptions and testing them.

Homework 5 was meant to build upon the previous homework by reusing the files library they built. The objectives of this homework is to teach students data-driven unit testing, and how to use files to drive tests. The homework also involved teaching JUnit 4 technique called parameterized test classes. A major problem with this particular homework was the lack of documentation about JUnit 4. For this reason I created an example in the homework about this feature. Unfortunately that was not enough, several students told us they had difficulty understanding the technique. So we had to extend the homework deadline and make a small lecture about it.

Grading of homeworks was very time consuming and tedious task. It took on average well over an hour per student homework to assign a grade with reasonable feedback. The worse a homework is, the more time it took to grade. That's because bad homeworks are always harder to understand. Even worse, sometimes debugging is needed to follow the code and decrypt its fuzziness. Moreover, with weak homework a longer and more detailed feedback is needed. It was not uncommon for some homeworks to take 2 or 3 hours to grade.

Final Exam

The final exam is a take-home exam. The first draft was distributed to the students about six weeks before the deadline. And the final version was posted four weeks before the deadline. The exam of 2009 class was a tuned down version of the 2008 exam. Two of the seven requirements sections of the 2008 version were dropped. Also, some other minor changes in the text were made in order to clarify some ambiguities. In 2008 no student was able to complete the exam requirements. It was not only because of the size of the exam, but also because all students, who had obviously underestimated it, started late (2-3 days before deadline).

Table 1 shows the 16 students who submitted their final exam. As we can see from the table, most students (especially those who did well) wrote on average between 25 and 35 non-blank lines of source code. There are outliers however. Student S10 for example submitted good exam but the evolution in code was weak as big chunks of code were checked-in though 19 iterations. It is worth noting that the exam had 22 distinct requirements, and most of them were too complex to be solved in one iteration.

Table 1 list of students who submitted their final exam along with the score, number of SVN commits, # of Lines of Code, and average #LOC added per commit

Student (score%)	S1 (98)	S2 (98)	S3 (95)	S4 (94)	S5 (90)	S6 (88)	S7 (86)	S8 (81)	S9 (80)	S10 (79)	S11 (52)	S12 (52)	S13 (52)	S14 (30)	S15 (28)	S16 (16)
Commits	83	52	50	116	61	28	34	105	33	19	21	21	73	57	18	8
LOC (non-blank)	3143	1996	1785	3567	1600	999	774	1660	1080	1429	884	639	415	918	385	489
LOC / Commit	37.87	38.38	35.7	30.75	26.23	35.68	22.76	15.81	32.73	75.21	42.1	30.43	5.68	16.11	21.39	61.13

In 2009, the students started early in comparison with year before. Two students did not submit the final exam, mainly because they had very low scores throughout the course and had little chance to make up. Figure 1 shows three students who started the earliest ended scoring in the top five. However, they started with a slow progress that increased as the deadline was getting closer. About 65% of the work, estimated by number of check-ins and LOC, was done in the last 72 hours before deadline. On the other hand, Figure (3) show students who got the lowest scores (bottom six). They started between 24 and 72 hours before deadline, with an exception of one student who started 7 days before deadline but still got low score.

The deadline was originally set to be by the end of December 5. But in the evening of that day students got extension of 6 more hours till 6 am next morning. Looking at figure (4) we see that actually few students used the extension to keep working. Only two students kept working till the last hour before the final deadline even though they seemed to be the least two students to need to work so late, if we look at the size of their exam and completeness.

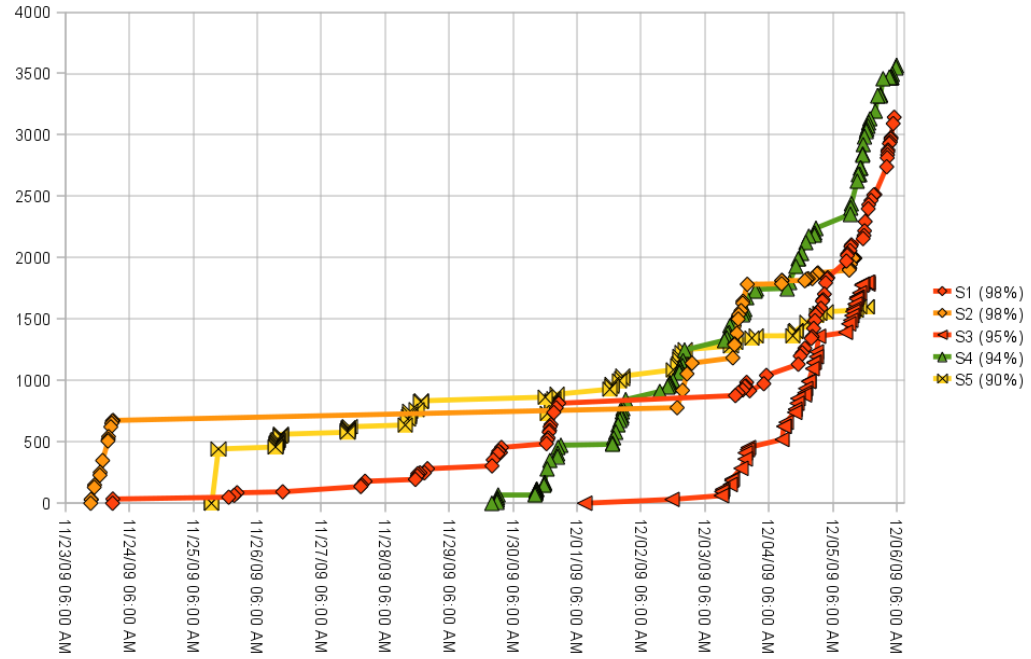

Figure 1 The exam code evolution, measured in (non-blank) Lines of Code, of the top 5 students

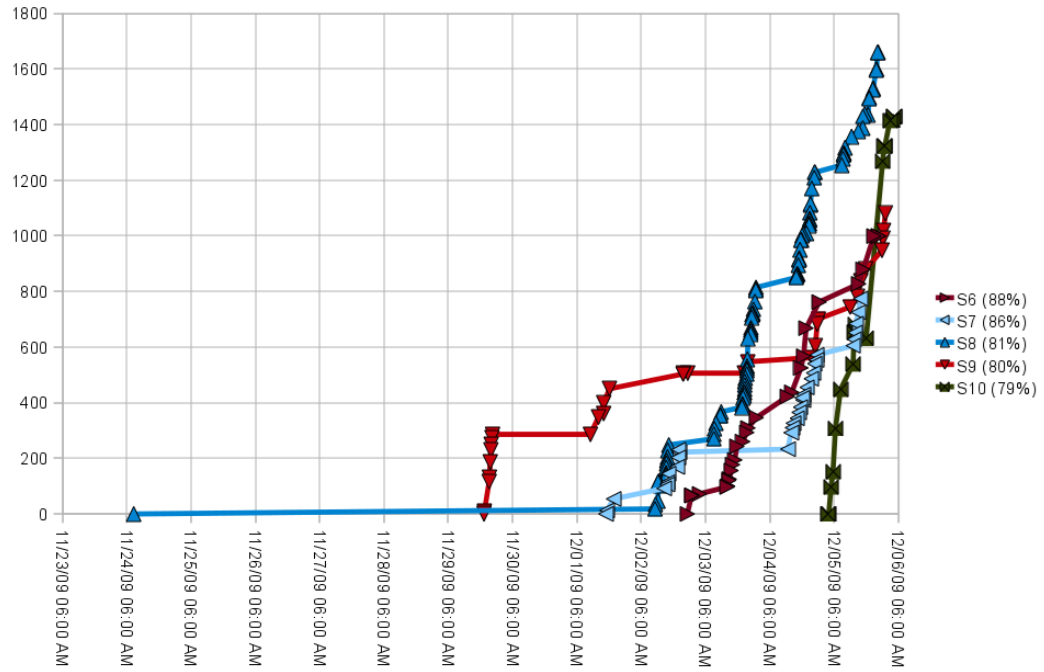

Figure 2 The exam code evolution, measured in (non-blank) Lines of Code, of the middle 5 students

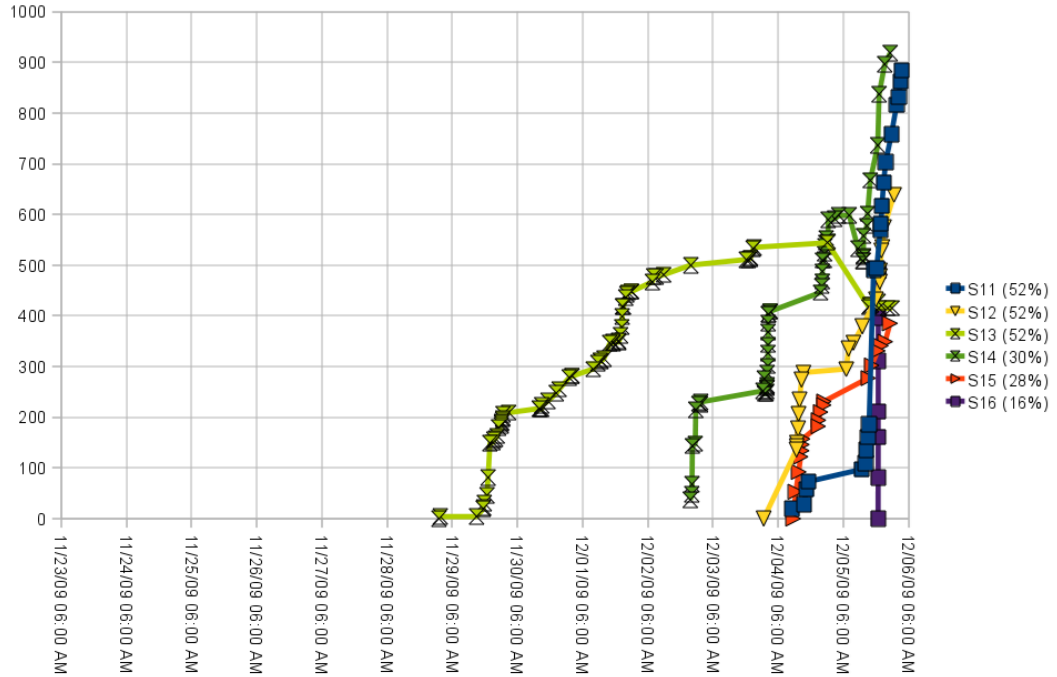

Figure 3 The exam code evolution, measured in (non-blank) Lines of Code, of the bottom 6 students

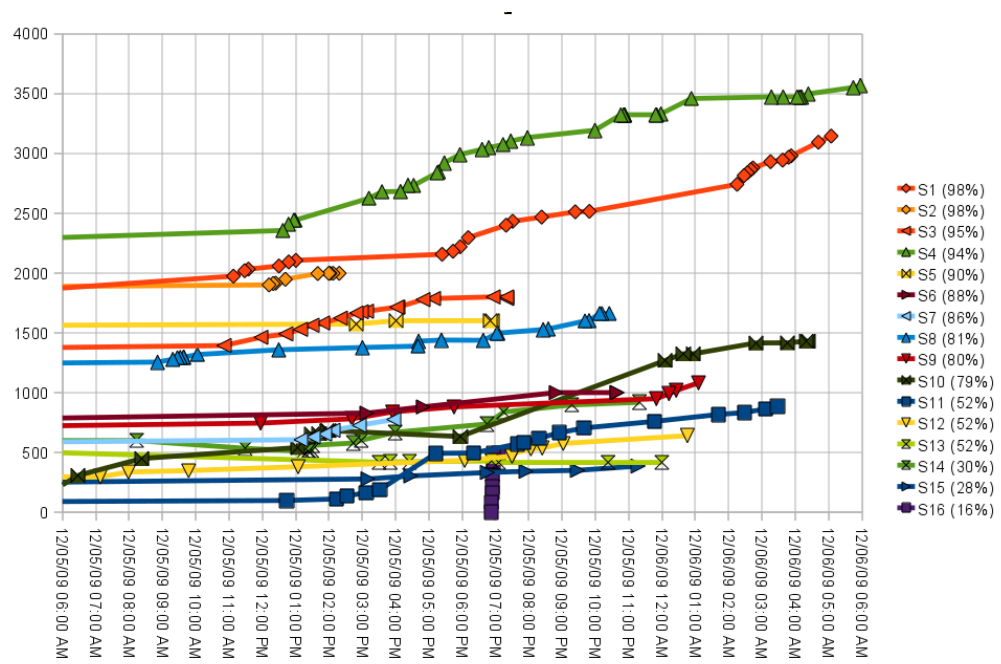

Figure 4 All students check-ins in the last 24 hours before deadline – which was extended from 12AM to 6AM

III. Common Implementation Errors

Writing "debug messages" to the standard output instead of depending on unit testing

This was a common mistake in the first two homeworks. Several students used a lot of printing to the standard output debug messages of kind "something wrong has happened" in the production code. Moreover, these messages were not tested. That is obviously a bad practice in unit testing, because you don't expect the tester to read the whole output log searching for anomalies every time the test suite is run.

Testing files I/O

This was actually surprisingly very common mistake in Homework 4 (see the appendix). Only 2 students (of 18) did it correctly. The other students chose to do superficial testing on methods such as MyFile.Delete() and MyFile.Exists(). The way they tested these methods was by checking the return value of these methods without checking the file-system to check whether the methods are actually doing their purpose correctly.

Hiding unexpected exceptions by using try/catch

This was another very common mistake. When writing a test for a method that is declared with "throws Exception", the java compiler will complain if you don't surround the calling statement with "try" block, or adding "throws" the test itself. Some students chose the first solution, without failing the test in case an exception happens. For example:

```
public void test() {
  try {
```

```
      foo("acceptable params");  /* foo is declared with "throws" but is
                                    not expected to throw any exception here. */
      . . .
   } catch (Exeption e) { /* nothing! */ }
}
```

This means that any exceptions that might unexpectedly happen, wouldn't be detected. In one case, the student wrote "e.printStackTrace()" in the catch block to print the exception to the standard error. This also means that JUnit won't detect the misbehavior and the tester has to keep an eye on the output. Whereas, two students wrote "fail()" inside the catch block. Although this case would at least trigger a failure, it is not a desirable approach because important exception information, such as stack trace and error message, are not passed to the tester through JUnit.

Misunderstanding of how exceptions work

An interesting and odd code pattern that was seen in the assignments of two students was surrounding a block of code that throws exceptions with try statement. For example:

```
public void foo(int x) {
  try {
     if (x < 0) throw new Exception();
  } catch (Exception e) {
     // Do nothing!!!
  }
}
```

It was not clear why some students wrote such code. For some, they were apparently trying to use exceptions as flow control statements. Anyway, it reflected misunderstanding of the purpose of exceptions and their usage.

Dependency on the order of tests

This is mistake was often non intentional. It happens when the code under test carries some state information between tests often because of static variables. In one case, a student has puzzled by the observation that one test only passes if another test was commented out. The reason was, of course, the unnecessary static variables in the class under test.

Avoiding magic numbers by using magic constants!

Since the checkstyle students had to use generated warnings at all magic numbers, some students chose to work around this by simply creating constants with names such as ZERO, ONE, TWO_THOUSAND_AND_NINE, and so on. This was the case with three students. Replacing magic literal with a constant named that way does not make it less magic than the literal.

IV. Resistance

Many students failed to use TDD. Some refused to program in this style. Most gave the appearance of compliance, but their work demonstrated either a lack of understanding or a lack of motivation.

Difficulty Decomposing Tasks

We tried the sorting example, but logistical problems in the lab turned this into an unhelpful exercise. We did learn that these students found it unnatural to imagine developing a sorting process by sorting 1 element, then 2 elements, then 3 elements, and then generalizing. They thought that the right way was to

solve the whole problem. We saw the same issue when they implemented heapsort for the midterm. Most students' sorting code came in en masse, with "iterations" revolving around testing and fixing, rather than being developed and checked into source control in ways that solved (and tested) smaller subtasks.

Size of Iterations

Decomposition was one aspect of a larger pattern—students checked relatively few iterations into the source control system, adding and testing large chunks of functionality at a time. This was more common problem in the midterm and homework than in the final. As apparently students learned from the feedback they got through the course to many small iterations. However, some students still had big chunks checked in for the final. Moreover, I noticed in two exams artificial commits, such as very minor changes (removing empty line or so) between check-ins.

Writing tests after the code, or not checking that the test actually fails with missing feature

The value of regression testing is that tests fail to indicate that possible bugs are introduced. This implies that test must be able to fail on some circumstances. However, some tests are written without being checked if they fail when the code is wrong. Students tend to ignore the fact that tests themselves should be tests. This is done in Test Driven Development by writing the test before the code and make sure it fails when the feature under tests is missing. This mistake was noticed in many assignments. A common code pattern that shows the symptom of this problem is something like:

```
public void test() {
  try {
    foo("invalid params"); // There must be a 'fail()' after this line
  }
  catch (Exception e) {
    assertEquals("Expected exception message", e.getMessage());
  }
}
```

Or even worse, an empty catch block was used in one assignment. The example above would fail if method "foo" throws an exception with a wrong message, but doesn't fail if the method doesn't throw any exceptions, which usually is the more important case.

Refactoring and Code Smells

Refactoring is an essential activity in agile development methodologies. Applying refactoring correctly and regularly ensures healthy gradual evolution of the project. While unit tests help ensure that refactoring is done properly without breaking anything in the code. Incorporating refactoring in the course started as early as the third week, and continued throughout the course until the final exam.

In the third week, refactoring was first introduced to student as an in-class exercise (lab) to refactor their own code from the previous homework in order to facilitate changing the data structure they had used with another one. They were later asked to do refactoring in all their homework and assignments and mention it in the code comments or documentation. Unfortunately, it took more than what was expected for the student to realize that refactoring is not removing extra white-spaces and that it is more than just renaming a variable to make it conform to checkstyle.

As a result, it was decided to create two more labs about refactoring in which students were given code and were asked to name non-trivial code smells and show how to refactor it. They were asked to refer to

Fowler's classical book on refactoring [4] and Wake's "Refactoring Workbook"[5], in addition to web resources such as [6] and [7].

V. Conclusions & Recommendations

- Learning objects (videos) – once the assignments stabilize, it will be worth the investment to create step-by-step demonstrations of examples. (Note: Kaner's experience is that step-by-step in lecture often persuades students that X can be done, but students forget the details. A re-viewable demonstration will be better.)
- We need a different book. The lack of good textbook for TDD course is still a big issue. Until the appropriate book is created, we have three options:
    1. Try another programming book that teaches some TDD.
    2. Stay with the current book. In this case more work is needed integrate the book material in the course by adopting more of its exercises and tailoring for teaching of TDD.
    3. No textbook. This will require enormous effort and work to create the course materials.
- Creating supportive tools to aid in the assessment of students' works. As we've seen, grading students work requires a lot of effort and time, and is very tedious task. It involves a fair bit of mechanical tasks such as evaluating the code revisions, running the tests, and so on. Custom tools can be helpful to generate reports about the evolution of code by counting not only the lines of code over time, but also things like the number of tests, number of methods, etc.

VI. References

[1] L. Koskela, Test Driven: TDD and Acceptance TDD for Java Developers, Manning Publications, 2007.

[2] D.J. Barnes and M. Kolling, Objects First With Java: A Practical Introduction Using BlueJ, Prentice Hall, 2008.

[3] J. Langr, Agile Java(TM): Crafting Code with Test-Driven Development, Prentice Hall PTR, 2005.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999.

[5] W.C. Wake, Refactoring Workbook, Addison-Wesley Professional, 2003.

[6] Refactoring Catalogue, Available online: http://www.refactoring.com/catalog/

[7] A Taxonomy for "Bad Code Smells" , Available online: http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm

[8] ECL-Emma – Java Code Coverage for Eclipse, http://www.eclemma.org/

Appendix (A) - Script for reading and processing Subversion logs

I created this modest script for the purpose of extracting some useful information from the final exam code revisions about the project evolution throughout the time. Some of the analysis presented in this paper was based on the data generated by this tool. This script might be helpful for other people as well. It could be used to do a similar analysis on other courses, or even on a general programming project that uses subversion.

The script will iterates through each revision on a specific folder and generate a comma-delimited values (csv) file. The generated file contains the date and time of each revision and the total number of non-blank lines of source code. Other data, such as check-in comments, might be generated as well by the modification of the script.

The script is written in bash and should run without any problems in Unix-like machine that has subversion (svn) and awk installed. It wasn't tested on Windows, but it should work fine using cygwin.

```
#!/bin/bash

oldpath=`pwd`
if [ "$1" ]; then cd "$1"; fi

svn update

svn -v log | grep '^r[0-9]\+'|
while read line; do
  rev=`echo $line | awk "{print \\$1}"`
  time=`echo $line | awk "{print \\$5,\\$6}"`
  svn update -${REV}
  loc=`find . -name '*.java' -exec cat '{}' \; |grep -cve '^\W*$'`
  echo "$rev, $time, $loc" >> results.csv
done

cd "${oldpath}"
```

The script should run against an SVN working directory (i.e., directory that already has checked-out files using SVN). If no directory is specified, it will run against the current one. The script will go though all revisions that are relevant to the target folder and generate a csv file containing the revision number, the time, and the (non-blank) LOC . An example to run the script (assuming the script is called "svn_process.sh"):

```
$ ./svn_process.sh TARGET_DIRECTORY
```

And this will generate a file called "results.csv" in the target directory that could be opened by Openoffice.org Calc or by MS Excel. However, MS Excel doesn't automatically recognize the date-time field as a date-time type. Therefore, it might be better to put the date and time in separate fields by adding a comma (,).

Appendix (B) - Homeworks

I. HOMEWORK 1 – Hello World!

This task is intended to get you up and running with Eclipse and JUnit. You MUST develop in Eclipse, not BlueJ. Submit code for each iteration. I'll talk about the submit server on Tuesday and we'll send an email about it tomorrow.

Iteration 1: Create a Hello class with a helloWorld method that returns the string "Hello World". Execute the class by calling it from JUnit and test that you receive the string "Hello World" in JUnit. The program doesn't have to accept any keyboard input or print anything to any screen.

Iteration 2: Create a helloWold method that accepts a string parameter. If the string is null, return "Hello World." If the string is not null, return "Hello " plus the string. Check the result in JUnit. Run a suitable number of tests to be sure that the method behaves as it should under different inputs.

II. HOMEWORK 2 – The Ticket Machine

PLEASE POST YOUR WORK IN THE SVN REPOSITORY, AT https://*******/

This assignment parallels the TicketMachine example in the text. Please create the TicketMachine program, as described in the text, except that you don't need a user interface. Feed inputs to your TicketMachine class methods via jUnit and check the results by returning them as values that jUnit checks. Create your program incrementally, filing at least one iteration for each of the specification items (sometimes called "stories" below). Remember to refactor your code before saving an iteration.

1. Return the price of a ticket
2. Return a running total of the number of tickets sold
3. Return a running total of the amount of money collected
4. Accept an amount of money (in cents, so a dollar and five cents is accepted as 105) for payment of a ticket
5. If the amount is too small, ask for more (and return how much more is due)
6. If the amount is correct, issue the ticket
7. If the amount is too large, issue the ticket and return the correct change, in cents
8. Add error handling to reject negative amounts
9. Add error handling to reject non-integer amounts
10. Allow for N different tickets, each with its own price. (Types can be identified with integers, eg type 1, 2, etc)
11. Modify the code for returning the price of a ticket so that you must specify the type of ticket in order to get the price for that ticket.
12. Modify your other code to require specification of the type of the ticket

13. Do something sensible if someone starts paying for one ticket and then makes a payment for a different ticket before completing the transaction for the first
14. Allow someone to cancel a not-yet-completed transaction and get a refund of the amount paid
15. Allow someone to give the ticket machine a ticket in order to get a refund
16. Cope with a valid request for a refund that asks for more money than the machine has collected
17. Set an opening balance so that the machine has money on hand at the start of the day, so that it can provide refunds even if it sells nothing that day Finally, write some notes (not more than a few paragraphs) describing some of the ways in which you refactored your code. I don't yet expect you to have a wide battery of techniques, but I expect you to make an effort to clean up your code as you go, and I expect you to do it consciously enough that you can tell me about it. File this document as a text file, with your code.

III. HOMEWORK 3 - Date Arithmetic

This assignment parallels the Clock example in the textbook. But instead of time, you'll be dealing with dates. Create your program incrementally, filing at least one iteration for each of the specification items below.

Remember to refactor your code before saving an iteration. Please put comments on your refactoring in the notes you submit when you check-in the code.

Throughout the assignment, use Java's built-in date functions as oracles, to generate expected results for your tests.

Somewhere in the assignment (in at least one place), use each of the operators &&, ||, !, and %.

1. Create a class to encapsulate date (Day, Month, Year). You're not allowed to use Java's date type except as an oracle to compare with in your test code.
2. Its interface should accept/return date values as three variables (day, month, year).
3. Allow checking whether a date is in a leap year.
4. Protect your methods from invalid dates.
5. It can calculate the difference of two dates as a number of days (could be negative number if the second date is after the first).
6. It can calculate the difference of two dates as a number of weeks and days (e.g. 1 week and 3 days in a 10-day period).
7. You can add D days to a date to get a new date.
8. You can add D day + W weeks + M months to get a new date.
9. Test with several values of (D,W,M) and explain why you chose each.

IV. HOMEWORK 4 - FILES MANIPULATION

Create your program incrementally, filing at least one iteration for each of the specification items below. Remember to refactor your code before saving an iteration. Please put comments on your refactoring in the notes you submit when you check-in the code.

You'll have to refer to java.io package documentation. Java API documentation (javadoc) can be viewed online or installed to your local machine. By default, Eclipse will show you the online docs if you press Ctrl+F2 after selecting the package, class, method, or variable which you want to get help about.

1. Create a utility class called MyFile. It will wrap a File object providing basic functionality to handle files. The constructor takes a filename as a parameter.
2. The file object should tell you whether the file it represents already exists in the filesystem. (See java.io.File)
3. MyFile.delete() should delete that file from the filesystem if it exists.
4. MyFile.readAll() returns a string representing the whole text in the file. (See FileReader and BufferedReader, other classes are also welcome!).
5. MyFile.readLine() reads one line at a time, each call will cause it to read the next line. It returns the line read. If end of file is reached, it returns null.
6. MyFile.isOpen() returns true if a file was opened for reading or writing. The file is opened automatically at any read or write call, if it wasn't already open.
7. MyFile.close() will close the file after it was opened.
8. Closing a closed file throws an exception.
9. MyFile.write(String) writes a String to a file. If the file doesn't exist, create a new one. If it exists, it appends to previously written text. (See BufferedWriter, and FileWriter).
10. MyFile.writeLine(String), appends a line to the file. (Similar to System.out.println).
11. Writing using an open MyFile instance that was used for reading should throw an exception. A file instance can be used for either reading or writing at the same time.

V. HOMEWORK 5 - DATA-DRIVEN TESTING

1. Introduction

Parametrized Test Classes

A parametrized test class is like a normal test class, but instead of hard-coding your test cases variable values, you can use a custom method to generate the test cases programatically. The test cases are stored in a table-like format as a collection of n-tuples. Each tuple is a one-dimensional array of type Object (Therefore you can pass any type of data). Each element of the array holds a value of one variable.

For example, let's say you want to test the factorial method. You have one input variable (int), and one output variable (int). So you need a list of test cases such as: {{0,1}, {1,1}, {2,2}, {3,6}, {4,24},...}

You need to do the following 3 steps to make “parametrize” your test class.

1. Annotate the test class with @RunWith(Parameterized.class)
   The @RunWith will make JUnit invoke the special class “Parameterized” as a test class instead of yours. However Parametrized class will run your test class by passing the “parameters”, or the collection of test data, to you tests cases.
2. Create a public static method that returns the data set (as Collection<Object[]>), and annotate this method with @Parameters.

3. Create a constructor for your test class that takes same number of parameters as the width of your data set (the size of each tuple). And assign these parameters to instance variables in your test class. You can use these variables in your tests.

Example:

```
@RunWith(Parameterized.class)
public class FactorialTest {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
                                    {0, 1}, {1, 1}, {2, 2}, {3, 6}, {4, 24} });
    }
    private int input;
    private int output;
    public FactorialTest(int _input, int _output) {
        input = _input;
        output = _output;
    }

      @Test
    public void test() {
        assertEquals(output, Factorial.calculate(input));
    }
}
```

Tip: If your tests cases have variable size of parameters, you can replace the single value multiple parameters with one parameter of type array or list. For example:

```
    @Parameters
    public static Collection<Object[][]> data() {
        return Arrays.asList(new Object[][][] {
                                    {{0, 1}}, {{1, 1}}, {{2, 2}}, {{3, 6}}, {{4, 24}} });
    }
    private int input;
    private int output;
    public FactorialTest(Object[] args) {
        if (args.length == 2) {
           input = (Integer)args[0];
           output = (Integer)args[1];
        }
    }
```

CSV files

A Comma-Separated Values file is a text file storing tabular set of data. Think of the data as an Excel worksheet. In fact Excel can happily import/export this kind of files. In this file each line represent a table row. And cell values are separated by commas. For this assignment assume there's no header line in a CSV file, ie. the first line is the first data row. If a cell value is a string it is double-quoted (“this is a string”), if there are not quotation then it's a numeric value - assume int for this assignment.

Each line can have a different number of fields! In other words, one line might have 5 cells, and the next one is empty!

2. The assignment

Create your program incrementally, filing at least one iteration for each of the specification items below. Remember to refactor your code before saving an iteration. Please put comments on your refactoring in the notes you submit when you check-in the code.

You'll have to use CSV files to store your test cases data. Therefore you need to develop a utility class to help you read this kind of files. You'll need to re-use the code you created in last assignment. Of course by "re-use" we don't mean "copy-and-paste" code fragments from that assignment. You should add last assignment class(es) to your project.

1. Create a class that can parse CSV files and extract data as a list of arrays. Meet the CSV specification above.
2. Throw an exception if the CSV file is not correctly formatted. (Try to think of all error cases).
3. Create a simple parametrized test class that reads its data from a defined csv file with the help of the CSV file reader you wrote. You can test a simple thing such as java square function. (don't forget to check in any data files you use).
4. Write a method that sorts an array of integers using bubble sort (google it). You can hard-code your tests in the beginning. You should use Arrays.sort method as your oracle.
5. Create a CSV file containing several test cases of different lengths for your sorting method. Each line in the csv file contains unsorted array of integers.