Test-infecting Future Software Engineers

Grigori Melnik
SAIT Polytechnic
Calgary, Alberta, Canada
grigori.melnik@sait.ca

Tell me, and I will forget.
Show me, and I may remember.
Involve me, and I will understand.

Confucius

Introduction

Since 2002 I have led an initiative to redesign the Software Engineering stream of the Bachelor of Applied Information Systems program at SAIT Polytechnic. The goal is to make future software engineers more thoughtful about software quality and more familiar with software testing techniques. In three years, the program went from no testing to test-driven development imperative.

Testing used to be just one of the modules in the Software Engineering Life Cycles and Standards course. It provided mainly theoretical coverage of different stages of a waterfall-like process, in which testing was close to the project end. Students only needed to create a test plan and discuss some quality characteristics. There was really no in-depth understanding of what those characteristics meant, and how they should be achieved and evaluated.

With the redesign of the program, we introduced a course on Software Testing and Maintenance. The objectives of the course are to provide a modern perspective on software testing and test-driven development, to practice various techniques extensively, and to explore and understand possibilities and limitations of each technique. The idea is to "test-infect" future software engineers to the point when they consider testing a norm, one of the factors of their professional hygiene.

The course is offered in the first semester (technically, fifth semester as this degree program builds on a two year diploma program). The course has evolved over the last 3 years. In fact, it is in the constant refactoring stage. Here's the latest roadmap of the course:

- testing objectives and strategies, oracles, and heuristics;
- specifying business rules as executable acceptance tests;
- exploratory testing (in the context of dealing with a legacy component; we use "legacy code" in a sense of any code that does not come with tests);
- domain testing;
- test-driven development, test patterns, test case smells (e.g. assertion roulette, fragile test, slow test, data sensitivity, interface sensitivity, test logic in app);
- code walkthroughs and test case reviews;
- localization testing (usually in the context of system adaptation to the French locale);
- UI testing, thin UIs; decision making - what to test and what not to test;
- performance testing, profiling, mock objects;
- auto-builds and continuous integration.

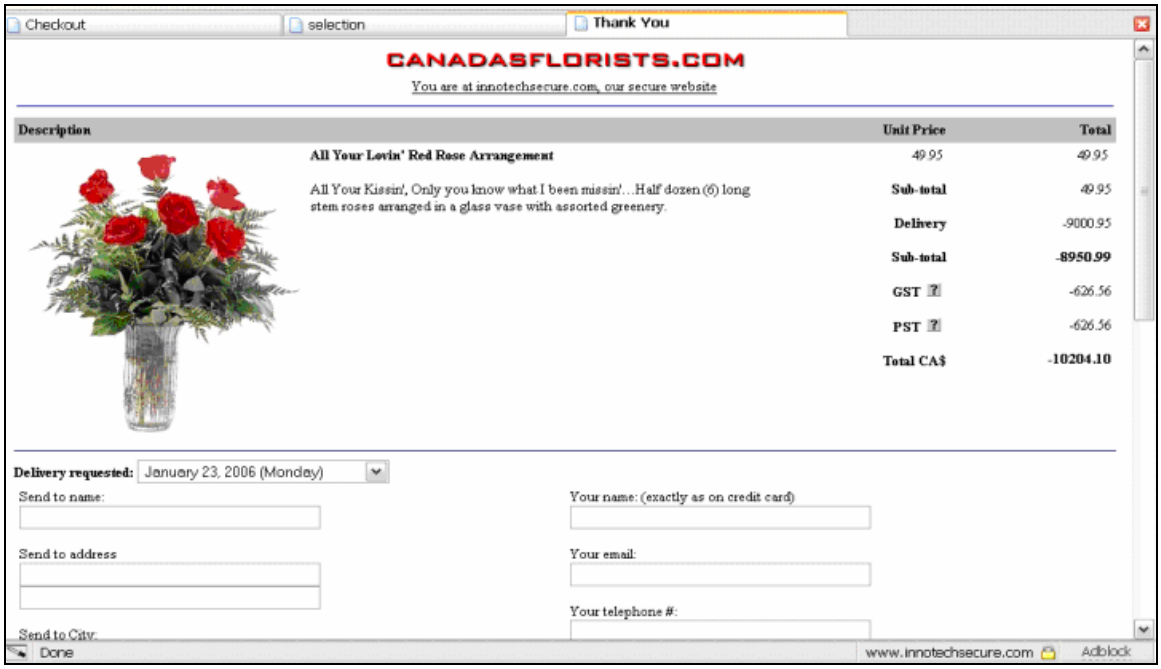
Following the principle of diverse half-measures, I make students recognize that no single technique can be used as a silver bullet, and the true power is in seeing which techniques can be effectively used and combined to achieve their objectives in a given context.

Setting the stage

The first thing I try to do in my class is wipe out from the students' heads the notion that software quality is someone else's responsibility (testers', quality control group's, project leader's, etc.)

I start off with a couple of "horror" stories (NASA Mars Lander, Denver airport automated baggage system, the US Vicennes, etc.) and then proceed with a demonstration of a serious bug. Security vulnerability exploits of e-commerce sites are quite good for this. Figure 1 shows one such demo. A web site initiates a transaction, but stores order particulars in hidden fields or cookies that can easily be tampered with. In this particular case, the site validates the unit price, but fails to validate the shipping cost. In a "gang" testing session, we were able to reduce it to $0.00 first, and then to negative $9000.95, which could have resulted in a loss to the owner if this transaction was actually submitted to and approved by the transaction clearing house. Though students may have read about software bugs and security breaches, seeing one exploited in front of their own eyes is a very powerful sobering experience.

Figure 1. Example of a Software Security Bug.



The demonstration is usually followed by a discussion of "good-enough software", defining quality (as "value to some person"), quality criteria (from both operational and developmental perspectives), testing (as a methodical "technical investigation done to expose quality-related information about the software under test"), and various testing objectives.

Games as course projects

Class assignments are designed as an iterative project (with 5-6 two-week iterations), with each iteration building on top of the previous ones. The project is usually some computer game. In the past, the games of Farkel, Rocket Mania, Bejeweled, and Big Money were implemented. On one hand, they are small enough to be completed in a single term, and on the other hand, they are big enough to teach various practices and not to overwhelm students who take 2 to 5 other courses at the same time. It is also easy to extend the requirements to an implementation of a networked, multi-player version. In the claim of Clark Abt that "a game is a particular way of looking at something, anything"[1], I find inspiration for my own approach to teaching about testing and test-driven development. A software quality consultant Elizabeth Hendrikson describes her experience with games as ways "to explore the way the world works in a small, contained setting.

Every game I play teaches me different lessons I can apply elsewhere in life. Games of chance teach me about risk. Games of skill teach me about strategies and tactics. Both kinds of lessons serve me well in software management." [3]

Team members rotate every two iterations. This way they get to read, test, and debug other people's code - which, I believe, is a very important competence for software engineers. For some students, it's a new experience since they are used to working with their own code only and they often suffer from the "not written here" syndrome.

Up until recently, I chose a new game for each term. However, I am considering an exercise of taking a game that had been constructed by students from previous semesters and giving it to the current students to maintain and extend. The idea is to expose students to other people's code with inherent smells and pitfalls. It's true that with pair rotations they get to work with someone else's code, however, that person is still available for questions and clarifications as she is attending the same class. In the case of maintaining the code written by the team that's already gone (which is common in the industry), students will face a bigger challenge, which would make them recognize and appreciate good tests and self-documenting code even more.

Here are examples of three activities to test-infect software engineers that I have experimented with. They worked well and were appreciated by the students (based on the informal and formal feedback collected during and after the course).

Bug hunts/races

Duration: 1 hour hunt + 1 hour debrief.

The instructor selects a piece of software and gives it to students to break. Students work in teams (of 2 or 4). The objective is to test the program to find bugs. The activity is strictly time-boxed (usually one hour). At the end, they submit their bug reports. When we reconvene as a whole class, we review the bug reports and give points (based on how good/serious each found bug is) and the team with the largest number of points wins the prize (usually, bonus points towards their grade). In this reflection stage, we try to uncover the way people were thinking when they found a bug. What prompted them to come up with a specific test case? What heuristic did they use?

Sometimes, I use an existing application and require students to test a particular set of features (e.g. a hex editor). In other cases, I write a small module and inject bugs in it (e.g date formatter in French, used for reporting). I've made an interesting observation about how students approach this task of finding bugs first. With the exception of one team, all teams attempted to decompile the application and figure out bugs based on the source code. Even though decompilation and disassembling techniques are not taught in this course, many students are quite familiar with them. It may be worth obfuscating the code first, if one wants students to practice black-box techniques.

Code puzzles

Duration: 5 mins individualized analysis + 5-10 mins discussion

The goal of this activity is to introduce students to the traps and pitfalls of a platform (Java, in our case) and to make them better understand counterintuitive and peculiar behaviors that often lead to bugs.

The instructor presents a code puzzle and a question (e.g. "What does this code segment produce?") The students are asked to think about it and to write down their answers. They should analyze the logics of the code without executing it on the computer. After 3 minutes, the question is complimented with a list of possible choices and the students are asked to analyze the code once again and see whether their original answer may change. After, volunteers offer their solutions and explanations of how they've arrived to it. We discuss different options till we get to the right one. This is when many students get that special 'aha' moment. We conclude with a

summary of the lessons learnt. Two examples of the code puzzles are given in Figures 2 and 3. Figure 2 deals with a silent integer overflow; while Figure 3 illustrates security vulnerability when the query string processed by a servlet is not validated, leaving the door open for a potential database exploit via an SQL injection. For additional ideas, I refer you to the collection of Java traps, pitfalls, and corner cases gathered by Java enthusiasts Joshua Bloch and Neal Gafter [2].

This activity is very short and can be used at any time during the session. It's a very good start to immediately involve students, and get them in the working and thinking mode. It is also useful in the middle of a session as an intermission (especially, in the long sessions).

Students always look forward to the code puzzle activity and are very eager to solve them. I found them valuable from both learning and entertaining perspectives.

Figure 2. Sample Code Puzzle: Silent Overflow Bug.

```
//What does the program print?

public class LongDivision {
        private static final long MILLIS_PER_DAY
                                  = 24 * 60 * 60 * 1000;
        private static final long MICROS_PER_DAY
                                  = 24 * 60 * 60 * 1000 * 1000;

        public static void main (String[] args) {
           System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
        }
}
```

Figure 3. Sample Code Puzzle: SQL Injection Vulnerability.

```
//What's wrong with this code snippet?

public void doGet(HttpServletRequest  request, HttpServletResponse response)
             throws ServletException, IOException {
        String userName = request.getParameter("n");
        Connection connection = null;
        String query = "SELECT * FROM Users " +
                       "WHERE name = '" + userName + "'";
        try {
           connection = DriverManager.getConnection(URL,USERNAME,PASS);
           Statement statement = connection.createStatement();
           ResultSet results = statement.execute(query);

           while (results.next()) {
                // fetch each row from the result set
           }
        }  catch (SQLException e) {
           e.printStackTrace(System.err);
        }
}
```

Code and test case reviews

Duration: 1 hour

I believe an important skill of a software engineer is the ability to read, inspect, and analyze other people's code. Code and test case reviews help to develop this skill. One of the teams volunteers their code base. We designate a reader (who is responsible for walking through the code and explaining what it does). Everyone else plays the role of an inspector. Done as a group, inspectors find various implementation issues (e.g., coding errors, missing test cases, non-conformance to coding conventions, poor coding style) and also question design decisions and suggest improvements. We found this activity effective when implemented both as an informal review and as a more formal inspection when the checklists were used.

Students found this activity to be extremely useful. And if at the beginning of the term, it was not easy to find a team to volunteer their code, we had no trouble finding them later. In fact, there were more teams willing to have their code and test suites reviewed than we could possibly handle in the time allotted.

These reviews allow for a wide variety of feedback from both the instructor and the peers. In addition to the benefit of improving code quality, there is another reason why I recommend code reviews to be adopted in as many software engineering courses as possible. Learning to give and accept criticism is important. Code and test case reviews allow for peer critique that helps students develop these skills in a controlled, non-threatening environment.

Conclusion

I am absolutely convinced that various software testing techniques must be taught and practiced by future software engineers. In order to "test-infect" them, we need to engage them in exciting projects. Activities outlined in this paper (implementing interactive games, bug hunts/races, code puzzles, and code/test case reviews) help me achieve just that. In addition, we, as educators, should emphasize the value of good test cases as project assets and require those to be submitted as part of deliverables in all software engineering disciplines where any kind of code is written or touched.

References

1. Abt, C. Serious Games. University Press of America: 2002.
2. Bloch, J., Gafter, N. Java Puzzlers, Addison-Wesley: 2005.
3. Hendrickson, E. "Thinking games". Online: http://www.stickyminds.com/sitewide.asp?ObjectId=3437&Function=DETAILBROWSE&ObjectType=COL