Test-Driven Development:
Does writing software backwards really improve quality?

by Grant Lammi

One of the biggest challenges of adopting Test-Driven Development (TDD) surprisingly is not technical. Instead it is often the psychological challenges that cause TDD to be used poorly or not adopted at all. For developers, writing in a TDD style can be akin to signing their name with the wrong hand. While it is certainly possible to make the signature look the same, it takes more time and concentration because it just feels unnatural. Furthermore, the idea of writing test code in order to exercise production code is already strange enough for most developers. The situation can then seem downright bizarre when coupled with writing test code BEFORE the application code.

Frequently, developers are also reluctant to accept a method that requires them to write more code than they did previously. This stems not only from the age-old programming adage that less code is better code, but also from a fear that the new method could jeopardize already tight schedules.

This article examines a project that used both TDD and traditional development methods. Many TDD projects have a large number of variables, which makes it difficult to successfully quantify the benefits. However, the project discussed in this article has a couple of advantages.

1. One developer worked on all the features for this release, which made the programmer's skills a constant and eliminated a large amount of variance.
2. This release was small in scope and was shipped with a larger release of another product. This took the stress of scheduling and the temptation to cut corners away since, even if using TDD took twice as long, the project would still be completed easily before the larger release.

The project was written in C++ and had to support Windows, Mac OS X, Linux, and Solaris. We used the UnitTest++ testing library for unit testing because it is cross-platform nature and easy to use. To make an informed decision about the effectiveness of TDD, the new release features were divided into two categories. The first set used traditional software development methods. Code was written first, then tested by the developer, and then handed over to QA for formal testing. The second set used TDD to design and test code before giving it to QA. To evaluate the methods we compared the number of defects found by QA for each feature and gathered some empirical observations from the developer about how he felt the code came out.

Defects Found

The QA department found 61 total defects across all features. This included defects for the newly added features as well as existing ones. It is possible that some of these defects existed in previous versions of the product.

ALL FEATURES	
Severity Level	Number of Defects
Critical	7
Major	8
Minor	3
Annoyances	10
Cosmetics/trivial	33
TOTAL	**61**

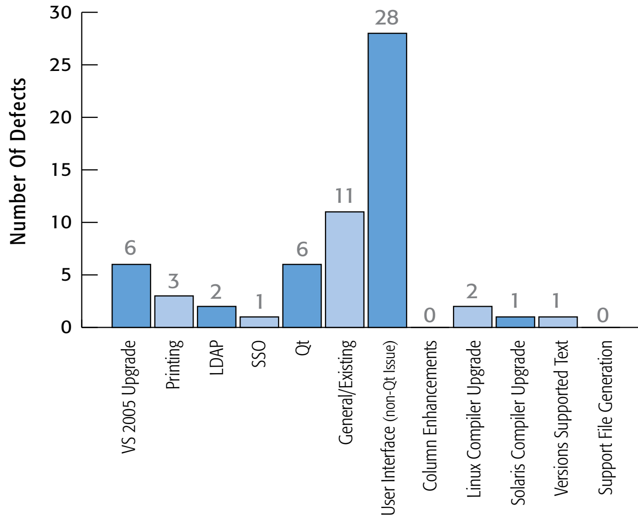
Focusing on only the new features, a total of 33 defects were found. When analyzing the severity, an increase in quality is clear, as evidenced by the lack of any Critical defects and a reduction in Major defects.

NEW FEATURES	
Severity Level	Number of Defects
Critical	0
Major	2
Minor	3
Annoyances	7
Cosmetics/trivial	21
TOTAL	**33**

Looking at the total number of defects broken down by feature, the data shows that most of the reported defects were either UI issues or upgrade issues to new libraries and compilers. (Qt, which is a

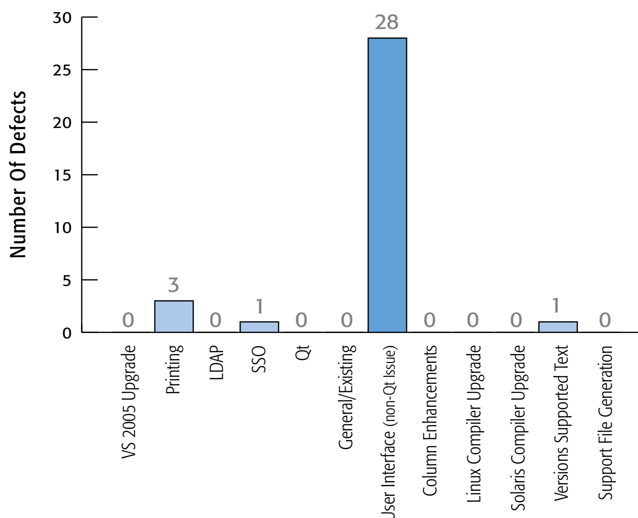
cross-platform UI framework, is used for the GUI and was upgraded to a new version in this release. The compilers for Windows, Linux, and Solaris were also upgraded, which caused some critical plug-in binary compatibility issues.)

Defects By All Features (63)



Removing the upgrade and existing issues from consideration and comparing only the new features provides a better view of the data. Notice there are no Column Enhancements or Support File Generation defects. The comparison also highlights how areas that are hard to unit test in general, such as the UI, are still problematic.

Defects By New Features (33)



When the defects from the new features are broken down into the traditional and TDD development methods, the difference in defects found is very apparent.

TRADITIONAL DEVELOPMENT	
Feature	**Number of Defects**
Critical	28
Major	3
Minor	1
TOTAL	**32**

TEST-DRIVEN DEVELOPMENT	
Feature	**Number of Defects**
SSO	1
Column Enhancements	0
Support File Generation	0
TOTAL	**1**

A final view of the data compares traditional and TDD methods, when looking at the severity levels of the reported defects. It is worth noting that the single TDD defect only occurred on a single QA test computer. The issue appeared to be an integration bug between a third-party library and an external system that only occurred under specific circumstances. It is entirely possible that it might not have been found if the QA department was not so thorough, which goes to show that no development methodology is a silver bullet to a quality.

TRADITIONAL DEVELOPMENT	
Feature	**Number of Defects**
Critical	0
Major	1
Minor	3
Annoyances	7
Cosmetic/trivial	21
TOTAL	**32**

TEST-DRIVEN DEVELOPMENT	
Feature	**Number of Defects**
Critical	0
Major	1
Minor	0
Annoyances	0
Cosmetic/trivial	0
TOTAL	**1**

Taking away cosmetic issues, which were not eligible for unit testing, there is still an 11 to 1 ratio of defects found when the two methods are compared.

Number of Tests Written

We wrote 45 individual unit tests for the features developed with TDD. The goal was not to have 100% test coverage but to create a solid design that would lend itself to easy updating. By not focusing solely on the number of test cases written, or requiring that tests be run on every official build, we made gains in technology areas that traditionally would not have been unit tested.

For example, the SSO (single sign-on) feature boils down to Kerberos integration. Kerberos is a secure authentication and authorization mechanism that is somewhat rigid in its client computer configuration requirements. Those requirements make it difficult to set up a process where Kerberos can be tested from any computer (e.g., usernames and passwords of Kerberos users would need to be stored where the unit tests could read them, creating a major security risk).

However, the design benefits that TDD brought to the other features were too good to pass up and we wrote the Kerberos feature in the same manner. We commented out its unit tests out once the code was checked into source control. The tests can be run when this area of functionality changes but they are not run as part of the standard build process.

It could be argued that all the Kerberos server functionality could be simulated using Fake or Mock objects but in the end that was determined to be too much extra work for a minimal amount of gain.

Empirical Quality of the Code

Code quality is always a touchy subject in the development world, with many different opinions on what is good and what is bad. According to the developer who wrote it, the code developed using TDD seems to be better than the code written in the traditional manner for the following reasons:

1. The TDD code was refactored several times during development. Methods were created, then changed, and then changed again with each revision, resulting in code that was tighter or clearer. The developer felt like the code went through an extensive proofreading cycle similar to what is done for newspapers, magazines, or the like.

2. There was a freedom of experimentation that led to better design decisions. Once the first working version of a class method was completed and properly tested, the developer could then try new design ideas. The unit tests quickly confirmed or denied that the new code worked, providing a digital safety net.

3. On a number of occasions, the developer found that the TDD method forced him to slow down and think more carefully about what he was trying to accomplish. Rather than just banging out code to get the job done, he focused on writing high quality code.

Conclusion

It is clear the TDD method resulted in fewer defects than the traditional method. The time difference between the two methods is difficult to measure because of the difference in features. However, an educated estimate has TDD development taking 15-20% longer than its traditional counterpart. It is certainly possible that this number will go down after the developer has more experience with TDD. However, it is worth mentioning it as a cost required for anyone trying TDD for the first time.

This project did not measure whether or not TDD leads to long term gains in quality. All indications suggest that TDD might be a compelling tool and it certainly warrants further investigation within your own development efforts.

About the Author

Grant Lammi is a technology evangelist at Seapine Software with over a decade of experience as a developer in the software industry.

5412 Courseview Dr., Suite 200
Mason, OH 45040

TEL 513-754-1655
FAX 513-754-1660

©2008 Seapine Software, Inc. Information presented here is accurate as of the time of printing, but is subject to change or revision without notice. 7190.3 Whitepaper Test Driven Development.indd 11/08