Software Testing: Study and Application in Piano Protégé

Tyler Barker

Abstract

A small study of software testing is presented, with an emphasis on regression and performance testing. The usage of such testing practices for a small project is discussed. The focus project is named Piano Protégé, a video game that simulates the task of playing the piano. The project was developed at Louisiana State University by a group of four for an undergraduate course in software engineering. The entire project, from requirements to testing, was completed within a semester of coursework.

1. Introduction to Piano Protégé

Piano Protégé is a project that borrows from the formula that recent popular games, Guitar Hero and Rock Band, have successfully used. The main difference, however, is the musical instrument that is emulated in the project. Instead of a guitar or drums, Piano Protégé, as its title indicates, allows a user to simulate playing the piano. This is achieved more realistically than in the two aforementioned games, because a user can connect a MIDI keyboard to the computer to use for the game. Therefore, while playing the game, a user is actually playing the piano, although the notes being played may not match the song's actual notes. In the game, the notes of a song are represented by rectangles that scroll down the screen in time to the music. When a note reaches the piano at the bottom of the screen, the user should play the corresponding note on the MIDI keyboard. If done successfully, the user's score will increase.

At the end of a song, a user can see his or her score and the percentage of notes correctly played. The notes that the game uses are represented by MIDI files that can be created in any MIDI editor. MIDI files contain many events that can represent when a note is pressed or released and provide information such as the volume and pitch of the note. Piano Protégé contains a MIDI parser that is only concerned with notes being pressed and what pitch these notes are at. When a song begins, the MIDI file is parsed, and the notes are stored in an ArrayList that contains the time and pitch of each note. As a song is playing, this ArrayList is traversed to determine when and where notes are to placed on the screen.

2. Introduction to Software Testing

Bertolino and Marchetti present a nice overview of software testing practices in general [1]. Testing techniques can generally be separated into two groups: static analysis and dynamic analysis. Static analysis does not involve the actual running of the software system. Instead, the code is reviewed for flaws and requirements are reviewed to ensure that specifications are being met. In dynamic analysis, the system is run, and the behavior of the software is observed for errors. Most studies in software testing focus mainly on dynamic analysis.

"Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior" [1]. In software testing terminology, a fault is a missing or incorrect piece of code. If this piece of code is activated, it creates an intermediate unstable state, called an error. If this error affects the output of the program, a failure can occur, which is the program's inability to perform its required action.

There are many types of tests that a piece of software goes through during its life cycle. Acceptance testing verifies that requirements are met. Installation testing verifies that the software can be installed on the target environment. In alpha testing, some in-house users try out the software to determine general flaws. In beta testing, alpha testing is expanded by deploying the system to external users. Regression testing ensures that changes to the software have not resulted in unintended errors. Performance testing measures things such as speed and memory usage. Usability testing evaluates the ease of using and learning to use the software.

There are also many levels of testing that a piece of software must go through. A unit test evaluates a very small portion of source code; usually the work of one programmer. Integration testing determines how these small units work together to create a larger component. A system test involves the entire software system.

In testing, many different tools are used to increase the effectiveness of tests. A test harness provides a controlled environment for test, allowing the results to be logged. Test generators create lists of tests for the software, whether they are strictly random or based on the software's specifications. An oracle is a tool used to determine whether the outcome of a test is successful or not. Tracers keep track of the history of a program's execution.

3. Testing Problems

Even though software testing can consume more that half of the software development cycle, testing practices are not as advanced as software development practices overall. Many people find more satisfaction in developing something new rather than testing someone else's work. "Many software engineers see testing as a junior or entry position and use it merely as a

springboard into development jobs" [2]. However, with the success of agile programming, testers have become integrated into the development process.

Also, there is a rift between academics and practitioners. New testing approaches have been researched by academia, but practitioners have been slow in applying these approaches. Practitioners also argue that academics are wasting time developing tools that are useless in practice. There needs to be better communication between these fields so that more technologies can be effectively developed and applied in industry.

4. Regression Testing

A piece of software is not a static entity. Changes are always being made, either to add functionality to the software or to fix an existing bug. When these changes occur, there is a good chance that there will be unintended side effects. This creates the need for regression testing. The process of regression testing attempts to ensure that a piece of software still correctly performs all of the functions it did prior to a change. In a world with unlimited resources and time, every change in software would be followed by a period of regression testing that runs every possible test case for the software. However, in the real world, this is not feasible; it is too costly and time-consuming. Therefore, a crucial part of regression testing is choosing the test cases that should be run after each modification.

When choosing test cases, many aspects need to be considered. The goal is to build a set of tests that is both safe and inclusive [3]. For a test set to be considered safe, it must include every test that could possibly find an error caused by the modification. An inclusive test set only includes tests than can uncover a fault caused by the change. Obviously, it is more important for a test set to be safe than inclusive. It is better to waste some time while finding every error than

to miss an error because of an incomplete test set. Using every test case that worked before the change is the easiest way to create a safe test set. However, it is a waste of time to retest aspects of the software that are completely unaffected by the modification, so the number of test cases can be reduced. The difficulty is reducing this set to make it more inclusive while keeping it safe.

Over the years, many different methods of selecting test cases have been developed. Most of these techniques involve looking directly at the code to determine which parts of a program will be affected by a change. Program slices can be created, which reduce the software to only the components that can be affected by a software modification. This program slice can be tested more efficiently that the program itself. Other methods involve looking at the settings and usages of modified variables, examining the reachability of modified code, and creating dependency graphs that determine which sections of the program are dependent on the modified code [4]. Empirical results from many previous studies show that although these test selection techniques can be cost-effective, there are many instances when they are not [5]. This can occur when more time is spent selecting the test cases than is saved by reducing the number of tests. Even worse, if a test selection method does not ensure that the test set is safe, an error can go unnoticed. This cost can easily outweigh the cost saved by the test selection technique.

5. Regression Testing in Piano Protégé

Fortunately, in our Piano Protégé project, it was feasible to rerun all test cases after making a major modification. This occurred a few times during its development. To start off the project, we first created an applet that played a song file while representing notes as rectangles scrolling down the screen. This worked fine, but it was not as fast as we would like. The

rendering of the graphics was occurring at less than ten frames per second. At this rate, the motion of the rectangles was very choppy. Also, in a music game, where timing is crucial, it is important that the graphics are rendered quickly. Therefore, we decided to use OpenGL (Open Graphics Library), a cross-platform API for 2D and 3D graphics. We accomplished this by using the JOGL (Java OpenGL) library. This significantly improved the performance of our graphics rendering. Instead of less than ten frames per second, we were able to achieve our goal of sixty frames per second.[1]

Theoretically, this change should not have affected how well the rest of our program worked. However, you can never take this for granted. Therefore, we performed a period of regression testing. We treated our program as a brand new program and tested everything that worked in the previous version.[2] Of course, nothing works as smoothly as you anticipate. Sometimes, while running the program, the rendering of the graphics would freeze, though the song would continue playing. We thought that this was happening because of deadlock from competing threads. To fix the issue, we needed to adjust the settings of the computer's video card. A problem with OpenGL is that its performance varies significantly depending on the video card that uses it. But the increase in performance was worth the potential problems it could create. Nevertheless, it was very important to do the regression testing before spending time on something else. Otherwise, the problem would have appeared sometime later, and it would have taken much more time to fix it then.

[1] Most video games made recently run at either thirty or sixty frames per second. Music games, like Guitar Hero and Rock Band, usually run at sixty frames per second since timing and responsiveness are so important. The graphics of Piano Protégé are not very robust, so achieving that frame rate was not a lofty goal. Attempting to reach an even higher frame rate is not very useful since many monitors cannot support this.

[2] This testing process is mostly explained later in the performance testing section. Any bug would most likely be found during performance testing. We were more concerned with how our major changes affected our game's performance. However, it was important to check that our game mechanics still worked. To do this, while playing a song, we played correct and incorrect notes and let notes go by without pressing them. We checked the scoring system to verify that all of the key presses were being registered and scored correctly.

We also had to perform a full period of regression testing when we decided to change the sound library[3] that we were using. The first library worked perfectly fine but lacked many features that needed. We could only start and stop a song file. Minim, the new sound library, allowed us to start, stop, and pause a song file and alter such values as the volume. Also, the time of a song could be accessed while it was playing, which allowed us to perfectly synchronize our graphics with the song we were playing. Once again, changing the sound library should not have affected the rest of the program, but we had to make sure. We started from scratch again, to test that everything still worked as efficiently as it did before the change. The only problems that we found were that songs took longer to load, and the playing of songs was more system intensive.

However, for most of the changes in Piano Protégé, it was not necessary to rerun all previous tests. The structure of our program allowed us to easily make program slices that isolated the modified code. For example, we created a class with the sole purpose of testing our parsing of MIDI files. Whenever we modified our MIDI parser, we would simply execute that single class, which would print output of the parsing. We could examine the output, and if the output was correct, we could safely assume that it would work when placed within the entire program. We performed similar tests for the code that read input from the MIDI keyboard. Finally, when changes were made to a single menu screen, it was not necessary to retest the entire program. We merely made sure that the modified screen looked correct and that the transitions to and from the menu screen still worked.

[3] A sound library is a set of predefined Java classes and methods that we used to play mp3 files. The individual songs we used were not affected by the sound library.

6. Performance Testing in Piano Protégé

Software performance testing is another very important aspect of testing, especially for a video game like Piano Protégé. The simplest way for us to test for performance was to merely run the program with a certain song playing. We could observe how smoothly the graphics were being rendered and we made sure that the notes represented on the screen were synchronized with the song playing. For more tangible testing, we calculated timing results while our program was running to ensure that we were getting the amount of frames per second that we want. We were able to set the number of frames per second that we wanted in our program, but if the program was not efficient enough, then the actual number would be lower. Therefore, it was important that we constantly tested this to make sure our program was efficient.

Also, if the program was not performing as smoothly as we wanted it to, we could test specific methods to determine which ones were consuming the most time. To do this, we could print timing numbers before and after the method was called to calculate how long the method lasted. For example, at one point in the program's development, we noticed that drawing the two piano keyboards at the bottom of the screen was taking more time than we thought it should. It probably did not make a real significant difference, but we decided to optimize the code anyway. Instead of drawing the two keyboards, rectangle by rectangle, to the screen at every frame, we drew one keyboard to a buffered image at the beginning of the program's execution. Then for each frame, that single image was painted to the screen. This process decreased the number of CPU cycles needed for each iteration of the program's graphical loop.

Some types of software performance testing include load testing, stress testing, and endurance testing [6]. In load testing, a program is tested under the program's expected load. For our project, this involved testing using a song of normal length with an average number of

notes. Stress testing involves trying the break the program by giving it an excessive load. The best way for us to perform stress testing was to create a large amount of notes that needed to be drawn on the screen at the same time. We could test how many notes it would take to notice a decrease in our program's performance. We were able to force over thirty notes on the screen at one, which is significantly more than will ever be needed.[4] Finally, endurance testing involves running the program for a long period of time to test whether the program's performance decreases. This could be caused by such problems as memory leaking. The main method that we used for endurance testing was to leave our program running for a long period of time. We played multiple songs and traversed the menu structure multiple times to see if there was any noticeable effect on the performance. To test the actual game playing state of Piano Protégé for endurance, we made a very long song with notes located throughout. As the song played, we watched to see if there were any noticeable effects caused by the lengthy duration.

7. Conclusion

Both regression and performance testing are crucial elements to software development that should be repeated often throughout the process. Regression testing at each change will display problems immediately. That way, if a problem does arise, you will know which change caused the problem, and it will be much easier to undo this change. Regular performance testing can also save a lot of programming time. It may be tempting to keep adding features to a program; however, each of these features may keep decreasing the overall system performance. It you test for performance regularly, the degradation of performance will be noticed before

[4] Before using a MIDI keyboard to play Piano Protégé, we used the regular computer keyboard. While stress testing, we noticed that on many computers, the keyboard does not accept many three-combinations of keys being pressed at the same time. Thankfully, the MIDI keyboard does not suffer this problem, so we were not limited by how many notes a single chord can have.

adding more features. Then you can focus on increasing the efficiency of the current program before moving on to something else.

Luckily, the Piano Protégé project was on a very small scale. Therefore, testing the entire program was not very time consuming. At any point in the development process, we could decide to rerun all of our test cases, and it would only take a few minutes. Most research on software testing is concerned with the selection of test cases. The idea is to test as few cases as possible while ensuring that errors will be found. The only method that we used to make our testing process more efficient was to make simple program slices that isolated a major function of the program. Using more advance testing techniques would not have been cost-effective in our case; the testing method would have saved us less time than we spent developing it. Nevertheless, no matter how large a program is, testing plays a crucial role in its development. Piano Protégé was not an exception. Our testing methods were successful in that no major errors showed up late in our development process. This would have given us a big setback, making it hard to meet our deadline. Because of testing, errors were found and fixed in due time so that the development process could move along at a steady pace.

8. References

[1] Bertolino, A.; Marchetti, E., "A Brief Essay on Software Testing," Technical report, 2004

[2] Juristo, N; Moreno, A. M.; Strigel, W., "Guest Editors' Introduction: Software Testing Practices in Industry," *IEEE Softw.* 23, 4, pp.19-21, Jul. 2006

[3] Gallagher, K.; Hall, T.; Black, S., "Reducing Regression Test Size by Exclusion," *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on* , pp.154-163, 2-5 Oct. 2007

[4] Wei-Tek Tsai; Xiaoying Bai; Paul, R.; Lian Yu, "Scenario-based functional regression testing," *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International* , pp.496-501, 2001

[5] Harrold, M.J.; Rosenblum, D.; Rothermel, G.; Weyuker, E., "Empirical studies of a prediction model for regression test selection," *Software Engineering, IEEE Transactions on* , vol.27, no.3, pp.248-263, Mar. 2001

[6] Meier, J.D.; Farre, C.; Bansode, P.; Barber, S.; Rea, D., "Types of Performance Testing," *Performance Testing Guidance for Web Applications*, Microsoft Corporation, Sept. 2007, http://msdn.microsoft.com/en-us/library/bb924357.aspx