Integrating Performance Testing With the Software Development Process

by William J. Laboon

Managing Software Development
Final Paper
6 December 2011

Executive Summary
"Integrating Performance Testing With the Software Development Process" by W. J. Laboon

Software performance testing is defined as determining and analyzing the performance of a software project, including such aspects as the time it takes to perform operations, how many resources (e.g., CPU, memory, storage space) the system will use, and to how many users it can scale. Performance testing differs from program optimization, which is focused on improving the performance of a system. While performance issues are a common complaint by users of software, the performance testing process has not received a great deal of attention, formalization or recognition in the software engineering community. In this paper, the author describes a novel synthesized model of performance testing, called Incremental Performance Testing, which can be integrated into common software development methodologies, from classic Waterfall processes to Agile ones such as Scrum and XP.

Incremental Performance Testing consists of the following facets:

1. *Creating Proper Performance Requirements* - The performance testing team and relevant stakeholders should determine performance requirements which are quantifiable, testable, and specific. It is recommended that each performance requirement have a "target objective" and a "threshold objective," where a target objective is the goal and the threshold objective is the bare minimum. Performance requirements should be prioritized early on in the process.

2. *Modeling of Key Performance Indicators* - Even before coding has begun, it is possible to estimate the performance of the system, by using mathematical models along with baseline performance results of the proposed deployment hardware and any libraries being used. This can help find performance problems very early on in the development process, when they are easier and less expensive to fix.

3. *Preliminary Testing of Performance* - Although often done late in the development process, consistent performance testing of different sections of the project can find problems and possible problems much earlier on. There are several techniques to testing the performance of a system before the entire codebase is complete, including performance unit tests, feature tests, stub functions, emulators, and others. The performance trends of a system can be analyzed and provide information to stakeholders well before the final release of a project, so as to catch potential problems and estimate what sections will be problematic in their performance aspects at release. This section also details how to avoid premature optimization, with the key factor being to become more granular in fixes the further along in development one goes.

4. *Final Testing of Performance* - Before the software is released, the performance of the system should be tested with the user in mind, testing the system more formally, and with a broader scope, in a deployment environment. Whereas preliminary testing focused on providing rough estimates and technical analysis, this should answer the question, "How performant will a user find this system?"

5. *Integrating Performance Testing with Existing Processes* - Performance testing will be done much differently under an Agile model than a Waterfall model. In general, under more rigid, classic methodologies, more up-front work is done for determining relatively static performance requirements. More agile models will need less up-front work, but the resulting dynamic requirements inherent in agile development implies a more flexible performance testing model.

Incremental Performance Testing could be summed up as "Performance test early and performance test often." By integrating performance testing with the software development processes already in place, this can be done with a minimum of effort and resources. By determining where the performance problems are and where they are likely to arise in the system, stakeholders will have a better understanding of the final project and schedule. Project managers are less likely to be surprised by performance issues late in the development cycle. Implementing the process can help save time and resources, as well as improve the quality of the finished project.

Integrating Performance Testing With the Software Development Process

William J. Laboon, Graduate Student, Carnegie Mellon University

Abstract— **During the software development process, it is very common to leave performance testing until development is almost complete, when any found defects can be difficult to fix. Developers and testing personnel often do not understand how to properly test the performance of a software project until all functional requirements are met. Unfortunately, if problems are then found, it can be difficult to re-engineer the software to improve performance. It can also be difficult to understand the current performance of the system while development is underway. This paper proposes a novel process framework, Incremental Performance Testing, which is a synthesis of best practices in the performance testing field, and how to integrate this process with modern software engineering processes. It covers all phases of the software development process, from developing proper and testable performance requirements, designing and developing software with performance in mind, properly testing performance on incomplete software, and final performance acceptance testing. It will also outline strategies to focus on performance at the right parts of the development process while minimizing defects due to premature optimization.**

Index Terms— **performance analysis, software performance, software quality, performance evaluation**

I. Introduction

A. Definition of Performance Testing

Performance testing is the testing, measuring and analysis of the time and resources a given process requires in order to operate [1]. To the user, this could mean that there should be minimal perception of delays, that operations take a reasonable amount of time, and that the system degrades gracefully as load increases. To the system administrator, it could mean that the system is available and running, even under peak loads; that the program is not utilizing an undue amount of resources; and that mean and median response times are reasonable. To the developer, it can involve determining the proper algorithm to use, the trade-offs between various resources such as memory versus speed, or the optimization of code for a specific set of hardware.

To the performance tester, however, all of the above factors - and more - must be taken into account. It is the job of the performance testing team to determine what are minimal standards (with input from users and other stakeholders) for system response, usage patterns for system resources, and provide feedback to developers as they develop the project to meet these standards. This can entail a wide range of processes, from determining "wall clock" time via stopwatches or programmatically, determining system resource usage via performance monitoring tools, and analysis of algorithms, both empirically and theoretically. Additionally, whereas functional testing often has clear, Boolean results (either the system responds according to the way it was designed, or does not), performance testing results can often be fuzzy if the requirements are not extremely detailed, and an excessive amount of detail may make the requirement very inflexible. If a system can handle the required number of users at the cost of 100% of the CPU on a server, and there are other applications on the server which may need to run, did the system really meet the performance requirement? In some cases, yes; in others, certainly not.

As the majority of the software development processes move away from low-level programming and single-user systems, and towards broad-based usage over computer networks and accessing multiple libraries, subsystems, and APIs (Application Programming Interfaces), the concept of performance testing and analysis has moved from low-level optimization of assembly instructions to understanding the key performance factors of complex systems [2], [3]. For complex, networked systems, it is not only necessary to understand the performance characteristics of a given set of functionality, but also how these performance characteristics change under load, on different types of hardware, and with different network characteristics. As software becomes ever more complex, tools and methodologies more suited to the task of modern day performance testing need to be developed and used.

In the author's experience, there is a dearth of practical resources for today's performance tester [1], [4]. Whereas functional testing has been studied quite often, and there are a variety of specific papers on aspects of performance testing, there are few books or papers devoted to the broad study of software performance testing. In this paper, the author outlines a novel process framework for software performance testing, Incremental Performance Testing, based on both experience in the industry and a survey of current performance testing research. It also includes current problems in the field today that necessitate such a proposal; details on the proposed framework, including eliciting and determining requirements, automated and manual testing of the system's performance aspects, and working with the developers on improving system performance; and how to integrate this framework with existing software design methodologies.

[a] Manuscript submitted December 6, 2011.

W.J. Laboon is a graduate student studying for his Master's of Science in Information Technology, focusing on Software Design and Management, after spending eleven years as a professional software engineer at a variety of companies. He is a member of the Association for Computing Machinery and the Pittsburgh Technology Council. He may be reached at Carnegie Mellon University, 5000 Forbes Ave Pittsburgh, PA 15213 USA (e-mail: wlaboon@andrew.cmu.edu).

B. Current Problems in Performance Testing

Performance testing continues to be an under-studied aspect of system verification and validation [1], [4]. However, performance of software systems is a vital part of the usability of a system. A system may meet all functional requirements, but it is useless if users decide not to use it due to a perceived lack of performance.

One of the key issues facing testers of software with performance requirements is the belief that the software must be relatively complete and stable before running performance tests on it. On its face, this makes sense; how can one determine the performance of an application before it can run correctly? However, as will be explored in this paper, there are several ways of estimating aspects of final performance early in the software development process, even without a fully functioning system.

Another problem with many projects is the lack of proper performance testing requirements. Requirements need to be quantified before they can be formally tested, and results need to be quantified before trends can be examined. If a requirement states that a system is required to "be fast," what does this really mean? What is the appropriate response time? How many users should the system have when testing it? What hardware should the system be running on? A vague requirement such as this can never be passed or failed.

Finally, there are few resources for providing a comprehensive plan for performance testing of a software development project. Although there are numerous case studies [3], [5], [6]) and several attempts at building frameworks [7], [8] and broader outlines of the performance testing field [1], [4], there is certainly room for improvement and synthesis of all of these approaches. In addition, with a few exceptions such as Dobson's paper on integrating performance testing with Agile methodologies [9], these papers do not go into detail about how to integrate the process with current software methodologies.

There are other issues with the performance testing of software. Although performance testing, and software testing in general, are essential to the software development process, testing is often seen as a cul-de-sac where mediocre programmers end up [10]. In many organizations, performance testing is performed on an *ad hoc* basis once a problem is found, oftentimes in the field [1]. Many software projects do not even consider performance and other non-functional requirements (also known as the "-ility" requirements, such as scalability, reliability, etc.), or if there are requirements listed, do not have a plan for determining that they are met. Addressing every problem with performance testing is outside the scope of this paper, but it is the author's hope that providing a comprehensive review of how to integrate performance testing with a software development process can ameliorate some of these problems.

C. Overview of Proposed Solution

In this paper, the concept of Incremental Performance Testing is defined and explained. Just as functional defects are less expensive and easier to fix if found earlier in the software development process than later, determining early on if a system is likely to meet performance requirements can save development teams time and resources [4]. Additionally, the Pareto Principle is prevalent in the performance arena. Performance problems tend to skew, more than other defects, to a few key modules; according to E.J. Weyuker, 93% of performance problems are found in 30% of the subsystems of a project [4]. This makes it even more imperative to find out what these systems are, or might be, early in the development process. In order to find these performance defects earlier rather than later, this paper proposes a synthesis of some of the best practices of performance testing in a process known as Incremental Performance Testing. The key aspects of this are:

1. *Creating Proper Performance Requirements* - Requirements should be testable, specific and include all major factors necessary to determine the overall performance of the system. Although they should be influenced by qualitative reviews, the objectives should be quantitative. In most cases, there should be threshold requirements (absolute minimum for release of a project) and target requirements ("goal" requirements for developers). They should be focused on the performance aspects of the system that are deemed to be most important early on in the process.

2. *Modeling of Key Performance Indicators* - As mentioned above, a major problem with performance testing is that it is difficult to do testing of the entire system until a stable codebase is reached. However, using modeling techniques, performance bottlenecks and areas of likely performance problems can be estimated early in the software development process.

3. *Preliminary Testing of Performance* - Performance testing should never wait until the end of the development process. Catching a performance problem early can reduce the time to fix the defect dramatically, possibly by an order of magnitude. Modular performance testing and unit performance tests can be used to find potential performance defects long before the final testing phases. Performance tests can and should be performed before the entire system is complete by using stub functions, emulation software, and mathematical modeling. Performance trends can be analyzed so as to understand the direction the software is heading from a performance perspective. However, care must be taken to avoid premature optimization and balance meeting the performance requirements with meeting the functional requirements of a system.

4. *Final Testing of Performance* - Before software is released, its performance characteristics should be tested thoroughly. Performance problems which may occur in the field can be determined, and baselines for the software on different hardware platforms can be ascertained. The scope for testing immediately prior to release will be greater than for the preliminary tests.

5. *Integrating Performance Testing with Existing Processes* - Undertaking performance testing will be very different depending on the underlying software development methodology used. Performance testing should not be grafted onto a software development process without modification, but rather integrated into the testing methodologies of these processes. Several popular processes will be examined as examples, including waterfall and Agile methodologies.

All of these factors will be examined along with further

description and examples of best practices.

II. Creating Proper Performance Requirements

A. Understanding the Performance Testing Environment

The prime goal of performance testing is to determine where performance problems lie in the system and whether or not the system meets the performance requirements agreed upon by the stakeholders. This seemingly tautological statement is in actuality quite profound; a performance testing team is not an optimization team. A tester's role is to measure, and in so doing, allow development to improve. Performance testers can analyze problems in order to determine bottlenecks, but the role of the tester is to find the problems, not to fix them. On a small team, a performance tester may also be a developer, but he or she should remember that these are two separate roles [11].

Although there is a great deal to be gained by working closely with developers, a tester *qua* tester must always remain independent. Management's goal is to get software out the door; while that may be the correct decision even when the system does not meet performance requirements, performance testers should always be granted autonomy to determine whether or not a system has met the requirements.

B. Determining Elements to Test

What performance means will vary from project to project. The people to keep in mind when determining which elements to test are end users of the system - the administrators of the system, the customer who buys it, and the user who uses it. The performance testing team should be able to translate the needs of the customer into quantitative, testable requirements. The team will also need to be able to transmit this information to a developer in such a way that the developer understands the problem and can optimize the code, as well as to other relevant stakeholders. Some stakeholders may not have a performance-oriented, or even technical background, and so communication skills are key to explaining any performance issues that may be found.

It is impossible to list all of the various aspects of performance for which one may need to test. Among the factors one should consider are:

1. Wall clock time - This is perhaps the quintessential performance test. How long does it take for an operation to complete?
2. CPU Usage - How many CPU cycles were used to execute it? How much processor time was used?
3. Memory Usage - How much RAM was used? How much swapping occurred?
4. Storage Usage - How much space is being used on the hard drive by the application and its associated data? How much space is being used on different servers?
5. Operating System Usage - How many processes have been spawned? How many threads? How many system calls is the application making?
6. Network Usage - How many bytes are being transmitted? How many bytes are being received? How quickly are they being transmitted? Is there a point where network bandwidth or latency limitations inhibit the activity of the system?
7. Scalability - How performant is the system with more simultaneous users, more data stored, or more functionality being activated? Are there specific thresholds beyond which performance degrades noticeably? Where are the choke points?
8. Availability - What percentage of the time must the system be running? How much time is necessary for scheduled downtime? How much of the time is necessary to get the system up and running after an error occurs?
9. Throughput - How much data can the system process in a given amount of time? How many operations can the system perform in a given amount of time?

For some requirements, the tester will need to take a statistical measurement of multiple tests, such as the mean or median amount of time that it takes to complete. This will require multiple runs of the test, which should be taken into account when scheduling how long the testing will take. Also, transient effects, such as the startup of a JVM (Java Virtual Machine) or the loading of data into memory, may cause early results in a multiple-iteration test to return very different values than ones later in the test [12]. If this is the case, then the first n or n% of results can be eliminated or treated separately. The correct value for n can often be determined empirically, by looking at a chart of the length of time or amount of resources taken by the test plotted against time. Once determined, the value of n should not change when comparing performance across iterations.

The time and resources used by a test case may vary tremendously based on the inputs and the hardware being used. The tester should be sure to specify the hardware on which the test should be run. Input data should be specified; do not send in random data if the value of the data may influence the resources used. Consider plotting a location on a map. If the entire map is onscreen, plotting a low latitude/longitude value is the same as plotting a high latitude/longitude value. However, the time it takes to find the prime factorization of a number is highly dependent on the size of the number.

Depending on the scope of the application, providing an external interface to retrieve values from the system, or printing them out to log files, can be very helpful in determining sources of errors. For example, while the system-level monitor may be able to determine how many threads are running in a given process, the log file may show how many threads are accessing a database and how many are waiting for incoming network requests. This granularity can make it easier to determine trends and look for places to tune and optimize.

C. Determining Performance Testing Tools to Use

The modern performance tester needs to use testing tools besides a simple stopwatch (or its digital equivalent) [1]. At a minimum, he or she should have access to tools which allow the recording of system resource usage. For more complex applications, or ones which will have a large number of users,

database elements, or other parameters, automated scripting tools will allow performance testers to test aspects of the system which are impractical to test manually. It should be noted that the goal of this paper is not to evaluate the brands of tools, but to provide an overview of the types of performance testing tools which exist. Therefore, although particular examples of types of tools are given, these are not meant to be endorsements.

The *sine qua non* of a performance tester is a program to determine the resource usage of a system. This tool often comes standard on modern operating systems, although it may be necessary to purchase a COTS (Commercial Off-The-Shelf) tool or develop one on obscure or minimalist operating systems. These should be able to store data, not just watch it in real-time; watching resource usage with top (on Unix systems) or Task Manager (on Windows systems) and waiting for something bad to happen is not normally a good way to find performance issues. However, according to I. Molyneaux, this procedure, which he calls "watchful waiting", does occasionally prove useful in tracking down performance defects which only occur rarely or in a nondeterministic manner [1]. Even then, however, the performance results should be stored for later analysis.

Automated testing tools, such as the open-source Selenium or IBM's STAF/STAX, allow the tester to have a repeatable, complex sequence of events occur on the system, usually controlled by some sort of scripting or macro language. Since performance testing comparisons, in order to be relevant, should have as many variables as possible the same between iterations, the use of automation should be encouraged. No matter how well-written or well-designed a series of instructions for a user, there will occasionally be mistakes: a typo here, a mis-timed moues movement there. While automation can have a high up-front cost, it often saves testers time in the long run, due to reduction in human-caused errors, better quality and comparability of data, and reduced time for re-running tests [1]. Automated testing tools allow work to be done in the background while testers analyze data from other runs, write more test tools, or run even more tests.

Load generators such as Hewlett-Packard's LoadRunner provide a way to determine the behavior of a system under various loads. With a load generator, it is relatively simple to determine how the system under test scales with more users, more inputs, and more data. Since scalability is a key part of performance testing, load generation tools are used quite often [1].

External interface simulators will allow a system to be tested for performance even before these other subsystems are in place. It can also ensure that the test is only finding flaws in the system under test, and not in any external systems. For example, by having a simulator which always returns a random integer after ten seconds, the time and resources necessary for the operation to occur from the perspective of the original system can be calculated simply by removing that ten-second delay. It also allows performance testers to determine how extreme values from external systems could impact the performance of a system. If the system which normally returns a value in ten seconds suddenly starts taking a thousand seconds for some requests, a single-threaded system may grind to a halt, but a well-designed multi-threaded system may not have much of a user-facing impact at all. Simulators allow these extreme scenarios to be tested much more easily than forcing them with a working subsystem or faking them by such measures as pulling a network cable. However, it should always be kept in mind that these are in fact, simulators, and so should be seen as useful modeling tools, not as perfectly realistic.

Analysis tools can be as simple as a spreadsheet program such as Excel to view comma-delimited files of resource usage data, or an entire series of programs written by the test team. Be sure that the tools you use are able to handle the amount of data that you expect to generate. For example, earlier versions of Excel were unable to handle more than 65,000 rows of data. If you are taking resource statistics of a system once per second, it won't be very long before you are unable to fully analyze the data. If you do not have access to tools which allow you to handle the size of data you want, it may be easier to write scripts that minimize the amount of data, in essence "pre-processing it." To continue the previous example, the data could be reduced to 10% of its size by taking the arithmetic mean of every 10 seconds' worth of data, and creating a second file with those mean value. Note that this may cover up short spikes in resource usage in highly leptokurtic distributions. This processing of data is easily done with perl, bash shell scripting, or other scripting languages.

Keep in mind that performance testing teams will often have to perform the same tasks repeatedly. Using the correct tools will allow proper analysis of data, minimize human errors of transcription or other simple mistakes, and keep up team morale. It is much more efficient to use performance testers' time and energy on analysis of data than as stopwatch button pushers.

D. Creating Testable Performance Requirements

As with functional requirements, a performance requirement must be able to be tested and validated to be either passed or failed (or, in exceptional cases where the system cannot perform the task at all, marked as an error). However, even requirements which may sound reasonable at first may turn out to be unreasonable to test. Weyuker *et al*. mention a requirement that a compiler be able to compile any module within one second [4]. However, no matter how many modules are tested and found to compile within the specified time period, there always remains the chance that there is another module out there which would take longer than a second to compile. The requirement also does not take the hardware environment into consideration; does this apply to all systems that the compiler could conceivably run on, even horribly obsolete ones?

The following factors should be taken into account when designing performance requirements.

1. *Inputs* - All inputs should be specified either in the requirement, or refer to a specific set of inputs or how to acquire them. For example, a requirement may refer to a set of test data already in source control, or "fifty megabytes of random text in one table in the database." They should never require "a large database" or "text from an Internet forum."

Inputs should never reference “any” item unless the parameters and limitations of that item are obvious or also described. For example, “any valid latitude/longitude combination” would be valid, since the parameters of valid combinations are available. However, “any Internet comment” would not be a good data description, since there is no universal standard for length, character set, letter or word distribution, etc.

2. *Outputs* - The expected functional result should always be based, as much as possible, solely on the inputs. Performance requirements imply that the result being obtained is correct; a fast but wrong answer is definitely worse than a slow but correct one. In preliminary performance tests, the results need not always be correct, but the correct result should be known and specified.

3. *Environment* - The parameters of the hardware system on which the performance requirements will be tested should be specified in detail. For software projects which will run on multiple hardware environments, a variety of systems should be used to represent different performance specifications. For resource-constrained projects, virtualization may be used to simulate different systems, but the impacts of virtualization on performance should be kept in mind when using this method [1].

4. *Quantification* - The result of any performance test should be a number or set of numbers, with a specified system of units, and without subjectivity. Requirements stating that the interface should be “snappy” or the system should return results “within a reasonable period of time” are not testable. If a stakeholder insists on a qualitative requirement, it may be possible to transform it into a quantitative one by use of user surveys (a mean score of 4 or higher on a 1 - 5 scale where users rate the responsiveness of a system, for example). This will take a longer time to execute and is more subjective; it is given here only as a last resort and should be avoided if at all possible.

5. *Multi-Level* - Most requirements should include a target objective (what the minimum value would be under ideal circumstances) and a threshold objective. In some cases, this may be unnecessary, such as when the threshold objectives themselves are very aggressive and difficult to meet.

After applying these standards to the hypothetical requirement postulated by Weyuker in [4], referenced above, the problems with it are easily seen. A better performance requirement would be “On a system with four gigabytes of RAM and Intel Core i5-2435M 2.4 GHz processor, none of the modules under /src/test/TestModules/ should take longer than one second to compile, with a target of 800 milliseconds. All modules should pass the TestModule functional unit tests.”

Performance testers should exercise caution when including statistical requirements. Requiring that the median run-time of a test repeated 1,000 times be 200 milliseconds, for example, may hide some test cases where the test took several orders of magnitude more time. For tests being run multiple times, or tests run over a period of time, the maximum and minimum amount of resources used or time taken should be collected as well as mean, median, or mode measurements. For some tests, graphing the data out can be an easier method of determining the performance of a system. However, the requirements should not specify a graph, but specific numeric values [13]. Since any graph or relationship can be transcribed into functions or discrete values, this does not eliminate any testing possibilities, but does ensure that subjectivity is minimized by specifying more precise requirements.

It is important to keep in mind when designing performance requirements to plan for peaks, not for troughs [1]. System usage can vary tremendously, and one should always keep a worst-case scenario in mind. That being said, it could be very easy to go overboard on this, testing for a number of users several orders of magnitude than what management expects. Be pessimistic while still being realistic.

During execution of a test, there is always the possibility that a a system, especially one still under development, may crash or fail the functional requirements in some other way. If possible, make sure that the tests do not cause any unrecoverable damage to important data. If there is no need for a system to write data or delete data, then performance tests, especially long-running ones, should not do so, as a functional failure early on could cause much time to be wasted, as the rest of the test will be invalid [12] .

E. Prioritizing Performance Requirements

Early on in the development process, preferably during requirements elicitation, the performance requirements should be prioritized. A simple way to do this is by ranking; a requirement on resource usage may be the most important, whereas a timing requirement for a little-used feature of the software may be the least. This ranking will be subject to change over the course of development, especially under Agile methodologies. As the developers (and potentially the users) of the system gain a deeper understanding of the system, it is only natural for the performance priorities to change.

In an ideal world, where all requirements of a system are always met before release, and software developers always have enough time to optimize all aspects of a system under development, this would be unnecessary. However, in the real world, software may have its requirements cut in order to meet a deadline, features may be added which reduces the amount of time and resources for optimization, and sometimes projects simply run late. It is better to be prepared for this eventuality early in the process than trying to determine which aspects are more important when under intense time pressure, with a deadline looming in the near future.

III. Execution of Incremental Performance Testing

A. Overview of Incremental Performance Testing

The concept behind Incremental Performance Testing is to take the lessons learned from the advantages of incremental and iterative software development to the performance testing environment. Incremental performance testing stresses testing key performance indicators repeatedly and as early in the development process as possible. This is in stark contrast to traditional performance testing, which is often performed late in the development process, after the system has already reached a relatively stable point. At this time in the process, it

is often difficult to make major changes to the application without a concomitant major increase in the amount of resources necessary to do so.

Incremental Performance Testing involves modeling the system during the early design phases. This will necessarily be somewhat imprecise, but by modeling the system very early on, potential performance pitfalls may be seen even before coding has begun. Before official testing has begun, a baseline of the system or systems to be tested is taken. Unit tests will gather performance statistics even before a stable system is delivered to the testing team. As coding continues, each release is tested as much as it can be; if there are features that are not yet implemented, they are not tested, and if only part of the feature is complete, as much as can be is tested. Additional performance testing can be integrated into each release's feature tests. Throughout the process, the performance testing team interacts with project management and the developers in order to understand the prioritization of the performance issues found and possible solutions. Trends in the performance of the system are noted so as to understand which features and parts of the codebase are impacting the performance of the system, and to estimate the final performance results.

B. Performance Modeling

A key problem to performance testing is attempting to test performance without having a stable system. However, it is often possible to model the system partially or in full by using either stub functions or strictly mathematically. These can often provide a reasonable estimation for performance of the entire system early in the development process, but it should always be kept in mind that these are estimates and liable to change as development continues. Whenever possible, performance results from actual code should be used in place of models.

Modeling the system mathematically involves determining estimated usage patterns and how long the operations that the system performs will take [12]. By determining standard usage patterns and how long normal functions should take compared to how long they are taking, a delta can be obtained to determine whether or not the performance is trending correctly.

Using stub functions, one can simply estimate how long a function would normally take to return, then stub out that function with a delay and an arbitrary but reasonable return value if necessary. This should be done at a low level if possible. A designer may be more accurate determining how long a single database query would take than how long higher-level functionality may take, such as the entire login process. However, going too deeply into detail may break schedules. Running a performance test case normally will return a reasonable value for length of time, but note that this will substantially negatively skew resource usage. With the database query example, for example, a 500-millisecond sleeping thread will certainly read fewer bytes from storage than a real query.

In order to obtain the user data for modeling the system, capturing data directly from the user can be very useful, providing an accurate look at how users tend to use the system [14]. This data can then be played back in later performance testing for realistic usage scenarios. However, this will be more useful with systems with which users are already familiar, or ones similar to others with which they are familiar. If the user has no idea how the system works, he or she may have significant ramp-up time which is not a proxy for normal system usage (unless you plan to have many new users on a system on a regular basis).

C. Base Testing/Analysis of Hardware

Before any performance testing begins, the hardware on which the performance tests will run should be profiled and its specifications noted [1]. This hardware environment should remain unchanged as much as possible throughout the testing of the system. The performance of a system is very dependent on the underlying hardware specifications, much more so than with functional tests. Modifying the environment that a system is run on between builds means that the tester is no longer comparing apples to oranges. Note that this includes the software which is installed on the computer. For example, imagine a personal computer which includes only a base operating system, as compared to one that has numerous third-party software applications installed. Some of these applications may have processes running in the background, causing CPU usage and memory usage to vary. They will definitely modify the amount of space used on disk. These will impact the performance of the system and skew the results of any performance tests. One should remember to uninstall old versions of the software to be tested, and to install all necessary test tools (such as network analyzers, load generators, etc.) before any testing has occurred, in order to maintain a system in as similar a state as possible to previous tests. If time permits, or if the testing needs to be exceedingly accurate, the system can be wiped clean, and the operating system and necessary tools re-installed before each test. This may be difficult on a small project, but becomes a minor part of the process with proper disk imaging tools and relatively large projects.

In general, the performance testing layout should be as close to a deployment system as possible [1]. For software systems which will be deployed on a variety of systems with a wide range of hardware configurations, such as software for personal computers, several different systems with varying amounts of memory, disk space, processor speed, etc. should be used. If the performance test involves networking, the network should be isolated from other networks if possible so as to reduce interference from network load variance.

Finally, system statistics should be gathered as a baseline for the computer system, especially if the system is meant to be deployed on multiple operating systems. This can be as simple as running a performance monitoring program such perfmon (for Windows) or vmstat (for UNIX derivatives) on a system when not running any additional applications. This allows the base usage of the system to be corrected for, especially when comparing across operating systems. For example, a small, single-user operating system may have much less resource usage when an application is not being run, whereas a complex, multi-user operating system intended for a networked environment may have many background processes

running at all times, even when the system is not being used by a human being.

E. Unit Tests for Performance Analysis

Unit tests are a method of adding automated, code-based tests to check the functionality of individual "units" of test functionality, such as a specific class or method [15]. Performance data can be obtained from unit tests, allowing for rapid gathering of performance data with every compilation of the system [16]. By using tools such as JunitPerf, which can automatically add performance tests that can be run independently based on already-existing functional unit tests, developers can gain an understanding of the performance characteristics of their code with minimal effort [17].

Kim *et al*. in their paper "Performance Testing of Mobile Applications at the Unit Test Level" have developed a framework for performance unit testing for mobile devices, but which can be broadly applied to other software development efforts [16]. The concept is rather simple: create performance test scenarios which are executed as separate unit tests. Parts of the system which are incomplete or unavailable (such as connections to external systems) are emulated. Each performance test case has a maximum time allowed for it to execute. If it takes longer, it fails the test; if not, it passes. A profiler can be added in order to help determine the performance bottlenecks and hotspots, i.e., the parts of the code that are being called the most often.

Note that the results of unit testing may vary significantly from test to test if they are not done in a deployment environment. For example, if running on a separate build system, that server may operate much more slowly when numerous people are running builds in parallel as opposed to when there is only one unit test suite being performed. In order to minimize these issues and make relevant comparisons between builds, performance testers can make specific requirements as to system activity, such as locking the build system against other uses during execution of the unit tests. It would be wise to do this in an automated fashion during a time when other software developers would not need to use the system. Unit tests can provide a low-level and rapid way of determining current performance trends, but it should be remembered that they are not infallible.

F. Feature Tests for Performance Analysis

In many cases, already-planned functional test cases can be modified so as to gather performance data. By running an already-existing functional test case with a timer and/or a performance monitoring program, the performance of a particular feature across releases can be determined. This adds a minimal amount of work to the tester's responsibilities and can provide a finer granularity to the performance testing of the system.

Although this is a simple way to gather performance data, there are many faults from which it suffers. Feature tests are not developed to be performance tests, and therefore may give you faulty results. If they are done manually, there is room for human error in both timing and execution of the tests. In some organizations, functional and performance testers are separate groups, and so those testers executing these feature tests may not take into account the performance aspects of the system (e.g, by forgetting to close applications in the background or using servers with different hardware across test runs).

With these drawbacks noted, using feature tests to test performance is not recommended in general. However, for small or resource-starved projects, they may be one of the few ways to gather performance data for analysis during development.

G. Preliminary System Performance Tests

Although "vote early and vote often" [18] may be a poor way to run a democracy, "performance test early and performance test often" is an excellent way to run a software development project. Incremental Performance Testing attempts to test the performance of the system as often as possible and starting as early in the software development process as possible. This is in order to determine not only how well the software is performing at this moment, but to understand if the system is becoming more or less performant as time goes on.

During each iteration of the software, all features that have already been developed should be regression tested, as well as any new features that have come in with this release. As the software development process continues and more features are added, the preliminary performance tests will take a longer time to execute. Performance testers should take this into account when scheduling. With automated testing suites, however, this becomes more a problem of machine time and less of human time, which is, in general, more expensive.

The performance results should be gathered and stored, in order to see performance trends. These results should be kept under source control, test software control, or otherwise maintained. This allows the software development team to understand not only where they are, but where they've been, and their chances of reaching the system's performance goals. A system which shows an exponential decrease in performance at each iteration is a much starker wake-up call to project management than a poor result in one performance test.

In general, performance testing should be plan-driven. Sometimes, however, performance testers may do "exploratory testing" to do a broad overview of the system or check for specific issues which are brought up from development. When doing exploratory testing, the tester should continue to use best practices for performance testing, such as always gathering resource data. Exploratory testing should be seen as an informal way of gathering data, not as part of the standard preliminary system performance tests.

Before reporting performance results to stakeholders, they should be organized and consolidated. Most stakeholders will want to know the impact on project schedule and the user of the system under development, not specific resource usage patterns. Meier *et al*. provide an excellent list of items to keep in mind when reporting performance results: "Report early, report often; report visually; report intuitively; use the right statistics; and consolidate data correctly" [13]. When developing reports, try to think of what the stakeholders want to see, and let them know problems as soon as possible, as bad

news does not get better with time. The reports one provides should be easy for the non-performance tester to understand at a glance. It is also important not to mislead with statistics, and by understanding that not all stakeholders will have broad statistical knowledge. Charts and graphs, easily created with desktop spreadsheet software, are much better at showing performance trends than long lists of numbers. The manner in which reports are sent out can vary by project and process, ranging from a simple email from the performance team on a regular basis to official meetings.

I. Avoiding Premature Optimization

In 1974, Donald Knuth noted that "[w]e should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil" [2]. How does one reconcile this stricture against premature optimization with a call for very early performance testing of applications? The key difference is that optimizations based on the testing do not need to be made early on in the development of the system; it is often enough to know that a particular section of code will be problematic so that development time for optimization can be scheduled later. Additionally, understanding which modules and methods will be performance bottlenecks can allow the design of the system to be modified in order to minimize their use.

Another difference is that when Knuth was referring to premature optimization, he was discussing code-level changes; the primary discussion in his famous article was whether or not "go to" statements helped or hindered software development efforts at the time [2]. The optimization of designs, such as removing superfluous database calls, and the selection of more efficient algorithms, are certainly valid items for improvement relatively early on in the software development process. This differs greatly from modifying algorithms or low-level sections of code by hand. These features do not count as "small efficiencies;" using a O(1) algorithm as opposed to a O(n) algorithm (e.g., looking up data elements via a hash as opposed to a linked list) for large data sets can result in a significant performance increase with little risk of increased complexity. Understanding the performance characteristics of COTS/GOTS (Government Off-The-Shelf) libraries and other subsystems, and deciding which to use during the software design process, can also yield significant performance improvements [5]. Attempting to change the foundations of the system and libraries used late in the process would be a difficult undertaking.

During the preliminary phases of development, the focus should be on identifying potential risks. Numerous studies [5], [12], [19] have shown the efficacy of performance modeling and taking performance into account early in the software development process. As development continues, the focus can be shifted to implementing fixes to these risks. In general, the further along in development, the more granular the fixes should be. A failure for a library to meet a performance requirement, found during the design phase, may call for the replacement of the entire library; the failure of a particular test case during final requirement testing may involve modification to a few lines of code in a particular method.

IV. Final Performance Testing

A. Developing the Final Performance Acceptance Test

The final performance acceptance test is the last performance test before the software is released. At this point, numerous preliminary tests have been conducted, and the testing team is aware of which areas are likely to be weak, and which are likely to be strong. There should be a long history of what the performance trends have been, and a general idea of what the final performance will be like based on preliminary tests and system modeling. Additional data has been received from the performance aspects added onto unit tests and feature tests. Now, the system all has to work together in a deployment environment, or one as close to it as is feasible, for a complete performance inspection.

Whereas preliminary tests may focus on technical aspects of the system, the final test should focus on the system from the point of view of the end user and/or administrator. When developing the final performance test, the testing team should be thinking like a user and not a developer. As the guardian of the world of users from poorly-performing software, the tester needs to test the software against what a user would expect to encounter. It is the performance tester's job to translate what a user would think of performance into quantifiable aspects that can be tested, and translate that into advice for the software developers in a language that they understand. During preliminary testing, it is advisable to focus on testing aspects of the system which directly correlate to the codebase (e.g., testing how long a method takes to run). However, for final acceptance testing, the focus is on user-facing actions. Analysis of these user-facing tests may lead to checking how long a particular method takes to run, but this should be one step removed from the test itself.

When preparing the final performance acceptance test, it is imperative to keep in mind the amount of time it will take to execute all of the test cases and analyze all of the data. Final testing always takes place at the end of a release cycle when time is tight, and the time by which a performance testing team has stated that a build needs to get to them will often be violated as development runs long. This is why, based on the prioritization of performance requirements listed earlier, a prioritization of test cases which test those requirements should be set. If the release deadline for a software system cannot be pushed back, and less time is provided than was originally scheduled, then some test cases or parts of test cases will need to be cut. It is much easier to discuss this with project management if a plan is in place ahead of time as opposed to trying to cut items from the test plan on the fly. It also much easier to decide which elements to cut early in the process as opposed to the end of the schedule, under intense time (and probably managerial) pressure.

The final performance acceptance test should test all of the performance requirements that were determined early in the software development process and include a mapping from each test case to the requirement(s) it is supposed to test. Test plans should be reviewed by other testers, developers, and other stakeholders before execution to ensure that it tests the entire system and takes any additional factors necessary into

account.

Note that even though this is listed as the final acceptance test, it may be repeated several times. If there are elements which fail to meet the threshold performance requirements, developers may work on the system to fix the performance defects. In this case, the entire final performance acceptance test should be run again, if at all possible. It is tempting to simply re-run the failed test cases, but overall performance relies on many parts of a system. A fix to reduce the amount of memory used, for example, may increase the amount of CPU used, causing a different requirement to fail. During preliminary tests, this is much less important, as problems caused will often be found by other tests, or even go away on their own as development progresses. However, at this point, the performance testing team is the last redoubt against a poorly-performing system being released to the world. It pays to take the time to ensure that the entire system is performant.

During the creation of the final performance acceptance test, the tester should determine, with the assistance of other stakeholders, what should be done in the event of a test case or cases failing. Depending on the circumstances, there are a variety of possible responses. The system could be released as-is, re-worked to eliminate the performance problem, or the hardware requirements could be increased. The course to follow should be based on the prioritization of the performance requirement and the final result of the test case.

B. Elements of a Complete Performance Acceptance Test Report

After compiling the final performance acceptance test, the testing team should ensure that it contains all of the elements below [1]. A final report should be drawn up before release with a summary of all of these elements, although the depth that they will go into will differ depending on overarching software design process used and the technical prowess of the stakeholders to whom it will be reported.

1. A complete, precise description of the hardware environment in which the performance test was conducted.
2. A description of all scripts and test data to be used. This data should be under source control.
3. A mapping of the performance tests to the performance requirements which they test.
4. The version of software that was tested. This version should be under source control.
5. The test cases themselves, or a reference to them.
6. The results of each test case, and whether or not the performance indicator reached the threshold or target requirement.
7. If applicable, any additional gathered data from the test, such as CPU and network usage.

As with any test, functional or non-functional, repeatability is essential. As the results of performance tests often rely on many aspects of the underlying system, listing as much detail as possible on reports will allow future tests to compare results on an equivalent basis.

C. Dealing with Failure to Meet Threshold and/or Target Requirements

It is inevitable that at some point, on some project, certain aspects of the system may not meet even the threshold performance requirements. In order to minimize the fallout from this, planning for this eventuality should take place beforehand. Under ordinary circumstances, this should be done as part of the development of the final performance acceptance test plan. Some projects may be able to determine plans for this earlier, but due to the nature of the Incremental Performance Testing model proposed in this paper, the target and threshold objectives may not be known until later in the process. However, they should definitely be known by the time the final tests are being written.

The performance testing team should resist any temptation to reduce the threshold requirement in order to appease management or to “get it out the door.” This is the reason for threshold objectives, as opposed to target objectives: to provide a baseline for the minimum amount of performance that will be accepted by the customer. Modifying requirements because they are not met is antithetical to any structured testing methodology, but some managers are tempted to think of performance requirements as less important than functional requirements. The decision may be to release the software as-is, with the known performance defect, but the test cases should still be marked as failing for posterity's sake.

In all cases, it is better to let project management and other stakeholders know about problems and potential problems earlier rather than later. The earlier a problem is reported, the easier it will be to fix or to develop workarounds for it. Performance status reports or meetings can be very helpful in relaying information to relevant stakeholders, but the time and resources necessary for compiling these on a regular basis should be taken into account. Depending on the methodology used, these may be more or less formal and frequent. On a small Agile team, a weekly summary e-mail to relevant stakeholders may be sufficient; for a large team with a more heavyweight process, regular meetings with associated documentation may be called for.

V. Synthesis With Existing Methodologies

A. Towards an Integrated Performance Testing Plan

Performance testing is an integral part of the testing of the system, and thus an integral part of the development of a system. In a modern software development environment, there should not be any code “thrown over the wall” to the testing team, for either functional or non-functional requirements; rather, the testing team should be integrated with the developers [4], [9], for a variety of reasons. Both developers and testers will need to understand exactly what the performance requirements are and what they mean. Oftentimes, this is done more effectively via informal conversation than through precise documentation [9]. Although, as stated earlier, the tester's role is to find problems, not to fix them, understanding the software architecture and the development process allows the tester to create better tests.

With that being the case, the performance testing methodologies described above should be integrated with the existing software methodology being used by the organization,

not grafted onto it as an afterthought. Although the specific implementations of the concepts will differ depending on the already-existing software process, and on a company-by-company basis, the incremental performance testing model can be integrated with any of the popular software design methodologies today. The following sections will consider implementing the Incremental Performance Testing model with waterfall, Agile, and Test-Driven software development processes.

B. Considerations When Integrating Performance Testing with Waterfall and Heavyweight Models

Often considered the classic software development process, waterfall models provide a sequential series of steps, where the results from one step "fall" onto the next, from requirements, to design, to code, to test, to integration [15]. The output from one stage to the next tends to be relatively complete and well-documented. The traditional waterfall model has lessened in popularity, but various improvements upon the original design are very much in use today, such as the Incremental Model and the Spiral Model.

In some ways, this is the easiest model with which to integrate Incremental Performance Testing. The straightforward manner in which one goes from developing the performance requirements, to testing them for each iteration, and then performing a final performance test before release of the software. However, there are a few wrinkles in integrating it.

Under waterfall models, there will often be times where there is no recent stable release of the software. This is in stark contrast to Agile models where full releases, albeit with limited functionality, are made often. This problem can be worked around by using stub functions and simulators for the missing pieces of the software. There should be consistent testing of the modules of the software which can be tested, in order to provide a comprehensive picture of system performance trends.

Since so much more work is done up-front with these processes, the importance of modeling and simulators is increased. There is a longer period of time before a working system can be tested, compared to other software processes. Performance requirements will need to be spelled out in great detail, as compared to other methodologies.

With more iterative and incremental processes, such as Rational Unified Process and Spiral, these same steps will occur numerous times as the software undergoes revisions.

B. Considerations When Integrating Performance Testing with Agile Models

The Agile models of software development differ greatly from waterfall-based methodologies. The emphasis in the various Agile methodologies, such as XP and Scrum, is on close interaction with the customer/user; short, incremental releases and designs; lightweight processes involving limited paperwork and a belief that the code itself is better documentation; and being comfortable with change and prepared for it from the beginning of the project [15]. There are numerous benefits to using an agile process, but one disadvantage as it relates to performance testing is the necessity to change performance metrics and measurements as the project continues. The process of performance testing will be very different on a project where not only the underlying code, but the design and features of the project itself change from release to release. There are certain items which will be easier, and others which will be more difficult.

Since a key feature of agile methodologies is the release of working iterations on a relatively short time-line, performance testers can quickly test the performance of systems on a stable system. This solves a major difficulty in performance testing with other methodologies, since performance testers are often reluctant to test on systems that are not stable enough to produce a sufficient set of data [1]. Having stable systems released at short intervals means that there are definite, concrete iterations whose performance can be compared in order to understand performance trends. Instead of the "continuous" nature of some processes, where a testing team would have to deal with code frozen at some given point of time instead of an actual release *per se*, agile methodologies provide "discrete" releases meant to stand alone.

Changes in the basic design and features of the system, inherent in agile methodologies, mean that models of performance will be less useful. Since the features of the system, and thus what methods are likely to be called and in what order, are flexible, the models will have to be flexible as well. If the changes are significant enough, the time spent in building the models would be prohibitive. With the frequent releases inherent to an agile methodology, however, performance testing of the actual system, which is likely to be more accurate than a model, is much easier. It then behooves the performance tester to spend more time testing the incremental releases as opposed to depending on formalized models of the system.

Due to the fluidity of functional requirements in an agile environment, performance requirements must also be flexible. A benefit to using the agile process is that developers and testers alike can gauge the performance of the system by eliciting responses from the users of the software. Since agile methodologies produce working software early on, it is easier to get feedback on a system from potential users [9]. Even a developer with high domain knowledge may be surprised by what factors bother a user and what is seen as acceptable. Although there are numerous guidelines for human-computer interaction, and what counts as reasonable response times, these are not universal rules [1]. When interacting with users - especially those who are not themselves engineers or otherwise tech-savvy - it is up to the person eliciting the responses to attempt to quantify the statements of the user. Appropriate software tools (see Determining Performance Testing Tools To Use, above), potentially running in the background, can help avoid the almost necessarily qualitative statements given by users. For example, a user twice said that the system "feels slow," and the tester later notices that CPU usage at each of those times was above 95%. By contrast, the user mentioned the quick response times several times, all when CPU usage was under 80%. Based on these inputs, the tester can set a threshold requirement that CPU usage be under 95% during 95% of the system's expected usage, and a target requirement for it to be under 80%.

Although this flexibility can be a tremendous benefit to a project, it also places an extra burden on performance testing teams [9]. It has already been noted that performance testing is difficult to undertake when the system is unstable, either in features or the reliability of the software itself. Thus, for performance testing under an agile software development process, the use of lazy optimization is encouraged [20]. Lazy optimization follows the YAGNI ("You Ain't Gonna Need It") principle, allowing optimization to occur only after it is necessary to do so [20]. If problems are not found with the performance of a system, the optimization of that part is not done. Note, however, that this does not mean that performance analysis should not be done, or that rough estimates of performance should not be made early in the project. As stated earlier, performance analysis does not always need to be immediately accompanied by optimization. Understanding performance trends and areas for improvement can be helpful for determining implementation later in the development process.

C. Considerations When Integrating Performance Testing with the Test-Driven Development Models

Although an outgrowth of the Extreme Programming (XP) method of development, itself an Agile method, Test-Driven Development (TDD) has become an area of interest in its own right in recent years [21]. Many of the same issues faced during Agile development will be encountered in TDD, but there are several concepts specific to it which should be addressed. The concept behind TDD is to write the unit tests for a given software method, module, or other sub-unit of code first, before any development has begun [11]. Methods are stubbed out to return invalid values. Therefore, the unit tests should always fail at the beginning of the development of that method. As the method is fleshed out, more and more of the unit tests will pass, until all of them do and the method is theoretically finished.

TDD lends itself well to Incremental Performance Testing. Since unit tests are foremost in the development process, adding performance unit tests is natural extension. However, it should be noted that the output of a performance test should not be considered valid until all of the functional tests pass. Otherwise, all performance tests will trivially pass during the first iteration, as the method just returns a value; this will take up very few resources or very much time. This could mean that performance trends will be more difficult to deduce earlier on in development, as there will be a dearth of methods with acceptable unit tests from which the the tester can acquire performance statistics. In this case, models and stub functions should be used as substitutes.

VI. Conclusion

Despite there being concerns with computational power, resource usage, and the speed of programs since almost the invention of the computer, the field of performance testing still has room to formalization and expansion. However, with the Incremental Performance Testing model outlined above, software development teams can add rigorous performance testing to their existing methodologies in order to understand the performance of the system well in advance of its deployment, and to make any necessary changes at the earliest possible point in the software development process.

Although an entire process framework was laid out in this paper, development teams should feel comfortable picking and choosing the best and most applicable processes for their current project. Performance testing is not an "all-or-nothing" field, and some aspects of the Incremental Performance Testing model may not be necessary for all products. Software projects with very quick turnaround time, and little emphasis on performance, may have no need of a complex and rigorous modeling system. Other systems may have difficulty finding access to a deployment environment, such as embedded space systems, and so may spend much more time modeling performance than running tests on the actual hardware. The model is meant to be tailored to the development process already in place in order to enhance it, not as a straitjacket.

There are numerous avenues for further research in the field of performance testing and how it can be integrated successfully with current software development processes. The resource savings of finding a performance problem early versus finding functional problems early is an intriguing question. Determining the optimal number of performance testers and resources to use for a project of a given size is another. There is quite a bit of room to explore how open source performance is considered and tested, and how it differs from performance analysis in more traditional software development methodologies. While there have been some studies done of open-source performance software itself [22], there is surprisingly little information in the literature on how it is integrated with open-source development processes. An overview of how leaders of large open-source development projects determine whether their system is performant enough, or how individual programmers determine the performance of their own fix or feature, would be a beneficial addition to the literature.

The Incremental Performance Testing process explored in this paper is only an additional step towards the formalization of performance testing methodologies, and may be impractical for some projects and for some teams. However, it represents an improvement upon the current state of performance testing and analysis on many software develop projects today. Implementing this process can save time, effort, and resources, as well as allowing a better understanding of the current state of the performance a system under development.

References

[1] I. Molyneaux. *The Art of Application Performance Testing*. O'Reilly, 2009.

[2] D. Knuth. "Structured Programming With Go To Statements." ACM Journal Computing Surveys, Vol 6, No. 4, pp 261 - 300, Dec. 1974.

[3] X. Guo, X. Qiu, Y. Chen, and F. Tang. "Design and Implementation of Performance Testing Model for Web Service." 2010 2nd International Asia Conference on Informatics in Control, Automation, and Robotics. 2010.

[4] E.J. Weyuker and F. I. Vokolos. "Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study."

Transactions on Software Engineering, Volume 26, No. 12, pp. 1147 - 1156, December 2000.

[5] V. Liu, I. Gorton, and A. Fekete. "Design-Level Performance Prediction of Component-Based Applications." IEEE Transactions on Software Engineering, Vol. 31, No. 11, pp 928 -941, 2005.

[6] H. Kim, B. Choi, and W. E. Wong. "Performance Testing of Mobile Applications at the Unit Test Level." 2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement, pp 171 - 180, 2009.

[7] S. Chen, D. Moreland, S. Nepal, and J. Zic. "Yet Another Performance Testing Framework." 19th Australian Conference on Software Engineering, pp 170 - 179, 2009.

[8] C. Zhao, G. Ai, X. Yu, and X. Wang. "Research on Automated Testing Framework Based on Ontology and Multi-Agent." 2010 3rd International Symposium on Knowledge Acquisition and Modeling, pp 206 - 209, 2009.

[9] J. Dobson. "Performance Testing on an Agile Project." AGILE 2007, pp 351 - 358. 2007.

[10] E.J. Weyuker. "Clearing a Career Path for Software Testers", Software IEEE, vol 17, Issue 2, pp 76- 82, 2000.

[11] J. Thomas, M. Young, K. Brown and A. Glover. *Java Testing Patterns*. Wiley Publishing, Indianapolis, 2004.

[12] A. Avritzer and E.J. Weyuker. "The Role of Modeling in the Performance Testing of E-Commerce Applications." IEEE Transactions on Software Engineering, Volume 30, No. 12, pp 1072 - 1083, December 2004.

[13] J.D. Meier, C. Farre, P. Bansode, S. Baber, and D. Rea. *Performance Testing Guidance for Web Applications*. Microsoft Corporation, September 2007. Available online at: http://msdn.microsoft.com/en-us/library/bb924375.aspx

[14] A. Bertolini, , G. De Angelis, and A. Sabetta. "VCR: Virtual Capture and Replay for Performance Testing." Automated Software Engineering, 23rd IEEE/ACM International Conference on. pp399-402, 2008.

[15] F. Tsui and O. Karam. *Essentials of Software Engineering, Second Edition.* Jones and Bartlett Publishers. 2009.

[16] H. Kim, B. Choi, and W.E. Wong. "Performance Testing of Mobile Applications at the Unit Test Level." Third IEEE International Conference on Securre Software Integration and Reliability Improvement, pp 171-180, 2009.

[17] JunitPerf Home Page. http://clarkware.com/software/JUnitPerf.html. Retrieved 1 December 2011.

[18] Safire, William. *Safire's Political Dictionary*. Oxford University Press, 2008.

[19] J. Xia, Y. Ge, and C. K. Chang. "An Empirical Performance Study for Validating a Performance Analysis Approach: PSIM." Proceedings of the 29th Annual Internationa Computer Software and Applications Conference. 2005.

[20] C.-W. Ho, M. J. Johnson, L. Williams, and E. M. Maximilien. "On Agile Performance Requirements Specification and Testing." Proceedings of AGILE 2006 Conference. 2006.

[21] H. Erdogmus. "On the Effectiveness of Test-First Approach to Programming." Proceedings of the IEEE Transactions on Software Engineering 31 (1), 2005.

[22] L. Yuanyuan, X. Peng, and D. Wu. "The Method to Test Linux Software Performance." 2010 International Conference on Computer and Communication Technologies in Agriculture Engineering, Vol. 1, pp 420 - 423, 2010.

William J. Laboon received his Bachelor's of Science degree in computer science and political science at the University of Pittsburgh, in Pittsburgh, Pennsylvania, United States of America in 2001. He is currently studying for a Master's of Science in information technology, specializing in software design and management, at Carnegie Mellon University, also in Pittsburgh, Pennsylvania. His expected date of graduation is May 2012.

He was a Software Engineer for several years at Northrop Grumman Electronic Systems in Baltimore before becoming a Field Service Engineer, spending some time with systems in the field. He later worked as a Systems Engineer for the University of Pittsburgh Medical Center and as an Internet Application Specialist for Eyeflow LLC. He is currently employed with General Dynamics C4 Systems in Pittsburgh, although currently on educational leave.

Mr. Laboon is a member of the Association for Computing Machinery and the Pittsburgh Technology Council.