On Agile Performance Requirements Specification and Testing

Chih-Wei Ho[1], Michael J. Johnson[2], Laurie Williams[1], and E. Michael Maximilien[2]
[1]*Department of Computer Science, North Carolina State University*
{cho, lawilli3}@ncsu.edu
[2]*IBM Corporation*
{mjj, maxim}@us.ibm.com

Abstract

Underspecified performance requirements can cause performance issues in a software system. However, a complete, upfront analysis of a software system is difficult, and usually not desirable. We propose an evolutionary model for performance requirements specifications and corresponding validation testing. The principles of the model can be integrated into agile development methods. Using this approach, the performance requirements and test cases can be specified incrementally, without big upfront analysis. We also provide a post hoc examination of a development effort at IBM that had a high focus on performance requirements. The examination indicates that our evolutionary model can be used to specify performance requirements such that the level of detail is commensurate with the nature of the project. Additionally, the IBM experience indicates that test driven development-type validation testing corresponding to the model can be used to determine if performance objectives have been met.

1. Introduction

Agile development methods are sometimes criticized for not having explicit practices for eliciting non-functional requirements (NFRs) [13]. In this paper, we discuss an agile approach to address the specification and testing of an important NFR: performance. Performance is highly visible to the software user. A system that runs too slowly is likely to be rejected by all users. Failure to achieve some expected performance level might make the system unusable, and the project might fail or get cancelled if the system performance objective is not met [18].

Research shows that performance issues should be dealt with early, otherwise performance problems are more difficult and expensive to fix in later stages of the development process [6, 7, 17]. Performance engineering activities start with performance requirements (PRs) definition [10]. However, some of the performance factors are not predictable (e.g., an application may be run on an inferior processor with slower speed than initially anticipated) or not available at the early stages of development (e.g., the number of hits of a web server). To conduct a complete upfront analysis for performance issues is usually undesirable, especially for agile practitioners.

Therefore, we propose the software Performance Requirements Evolution Model (PREM) as a guideline for PRs specification and validation. Using the model, the development can start with quick, simple PRs and related performance test cases. During development, more precise and realistic performance characteristics are obtained through customer communication or performance model solving. The performance characteristics can be added to the PRs, and the test cases can be more detailed, until the PRs are "good enough" for the project and its functionalities.

In this paper, we provide an overview of PREM and a post hoc examination of a development effort at IBM that had a high focus on performance requirements. While software performance concerns include time (e.g., throughput, response time) and space (e.g., memory usage, storage usage), we limit and only focus on the discussion of time-related performance issues. However, we believe that our model is applicable to space-related performance issues.

The rest of the paper is organized as follows: Section 2 provides the background and related work for PRs and testing tools; Section 3 gives description of PREM; Section 4 presents an IBM experience with PRs specification and testing; and Section 5 concludes this paper with summary and future work.

---

2. Background

In this section, we present some background information and related work on agile-style software performance and performance testing tools.

2.1. Software performance and Agile Methods

At a glance, agile methods do not have explicit practices for eliciting overarching non-functional system properties. However, Auer and Beck list a family of software efficiency patterns called *Lazy Optimization* [1], which reflects the famous quote from Knuth that "Premature optimization is the root of all evil in programming,"[1] and the "You Aren't Gonna Need It (YAGNI)" philosophy. These patterns can be summarized as follows. Early in development, the system performance is estimated with a short performance assessment. Rough performance criteria are specified to show performance concerns in the system, and are evolved as the system matures. Tune performance only when the functionality works but does not pass the performance criteria.

Another camp claims that performance is mostly determined during the architecture stages [8]. Smith and Williams criticize that the "fix-it-later" attitude is one of the causes of performance failures [17]. Their standing ground is that performance models built in architecture and early phases can predict the system performance. Fixing software problems later is difficult and expensive. Therefore, performance decisions should be made as early as the architecture phase.

We believe the disagreement is rooted in the different philosophy of software development. For agile practitioners, software architecture is an instance of the big up-front design (BDUF) that is avoided. Lazy Optimization aims for agility, but may not be sufficient for high-criticality software projects, of which predictability is more desirable.

2.2. Performance testing tools

JUnit[2] is the most popular unit testing framework in the agile community. Beginning with Version 4 which was released in 2006, JUnit provides a "timeout" parameter to support performance-related testing. A test method with timeout parameter can only pass if the test is finished in the specified amount of time. Figure 1 shows an example of a test method with the timeout parameter. JUnitPerf[3] is a library of JUnit decorators that perform both timed and load tests. Figure 2 shows an example of JUnitPerf performance testing code.

```
//JUnit 4 timed test: only passes if it
//finishes in 200 ms.
@Test(timeout=200) public void perfTest() {
   //test scenario
   ...
}
```

Figure 1: JUnit timeout parameter

```
//JUnitPerf load test: run 10 instances
//of test, with 100 ms intervals between
//them - max elapsed time is 1000 ms.
Timer timer = new ConstantTimer(100);
Test test = new MyTest("Perf Test");
Test loadTest =
        new LoadTest(test, 10, timer);
Test timeLoadTest =
        new TimedTest(loadTest, 1000);
```

Figure 2: JUnitPerf example

These JUnit-based test frameworks provide an easy, programmatic way to write performance test cases. However, in complex test cases with complicated workloads, accurate probability distribution may be required [19]. JUnit-based test frameworks may be insufficient in such situations. Additionally, the resulting test code would be difficult to understand. To design more complicated performance test cases, one should consider more advanced performance testing tools, for example, script-based (e.g., The Grinder[4] [21]), user action recording (e.g., Apache JMeter[5]) or other commercial, high-end tools.

Agile approaches rely heavily on testing for software quality assurance [9]. Although, as stated in the previous section, Auer and Beck argue that performance concerns should be dealt with after the functionality is complete [1], we posit that a development team can benefit from early performance specification and testing.

3. Performance requirements evolution model

In this section, we provide an overview of PREM. PREM is an evolutionary model for PRs specification. PREM provides guidelines on the level of detail needed in a PR for development teams to specify the necessary performance characteristics details and the form of validation for the PR. Before explaining the model, a short example is provided to show how a

---

[1] "Computer Programming as an Art," 1974 Turing Award lecture.

[2] http://www.junit.org

[3] http://www.clarkware.com/software/JUnitPerf.html

[4] http://grinder.sourceforge.net/

[5] http://jakarta.apache.org/jmeter/

performance objective can fail because of underspecified PRs.

A PR might be as simple as "*The authentication process shall be complete in 0.2 seconds.*" An intuitive way to write a test case for this requirement is to run the authentication process several times, and compute the average. However, even if the average is below 0.2 seconds, when the system is live, and concurrent authentication requests come in, the users may experience more than 0.2 seconds of waiting time. In this example, the performance objective is not achieved because the workload characteristic is not specified in the requirement and not accounted for in the test case.

As the example illustrates, imprecise PR specification can lead to improper interpretation of performance objectives. Improper interpretation of PR may be more prevalent when agile methods are used because requirements, generally functional in nature, are documented informally in the form of stories and/or features.

3.1. Model description

PREM classifies PRs in four levels, starting from Level 0. Figure 3 shows the graphical presentation of PREM. To move a PR to a higher level, one needs to identify more factors that have impact on the performance of the system. However, PR refinement should stop at an appropriate level and should not be over-specified so that the detail of the PR is commensurate with the nature of the project and its requirements.

3.1.1. Level 0. Level 0 represents PRs with only qualitative, casual descriptions. Level 0 PRs bring out the operations for which performance matters in the eyes of the customer. Because of its informality, this type of requirement is easy to specify. Similar to stories in XP [3, 5], Level 0 PRs are the starting point from which customer and developer generate more precise specifications. One example of Level 0 PR is "*The authentication process shall complete before the user loses his or her patience.*" Level 0 PRs are usually specified qualitatively. As a result, they can usually be validated with qualitative evaluation. In the authentication process example, the development team might provide a prototype to the customer, and see whether the user is satisfied with the response time.

3.1.2. Level 1. Successful agile testing relies heavily on test automation [9]; human testing is undesirable and should be limited whenever possible. To make the requirement testable with automation, the first step is to provide quantitative expectations in the specification, and promote it to a Level 1 PR. The quantitative expectations might come from the customer, the development team's experience, domain experts, survey, or research.

Level 1 PRs are associated with quantitative metrics. Table 1 lists some typical performance metrics (adapted from [11]). For example, after some discussion with the customer, the developer can define a Level 1 PRs such as "*The authentication process shall complete in 0.2 seconds.*" Because the metrics are quantitatively measurable, one can run the authentication process and see whether it completes within the expected amount of time. A timed test that runs one particular functionality at a time suites this purpose well, and is very similar to how a single user uses the system. With proper runtime environment configuration, level 1 PRs may be sufficient for a single-user system.

Table 1. Typical performance metrics [11]

Type	Performance Metrics
Throughput	# of transactions / second # of messages / second # of pages rendered / second # of queries / second
Response Time	Transaction processing time Page rendering time Query processing time

3.1.3. Level 2. In commercial websites or multi-user systems, Level 1 PRs are insufficient because they do not show how different processes interact with each other and how they respond to system workloads

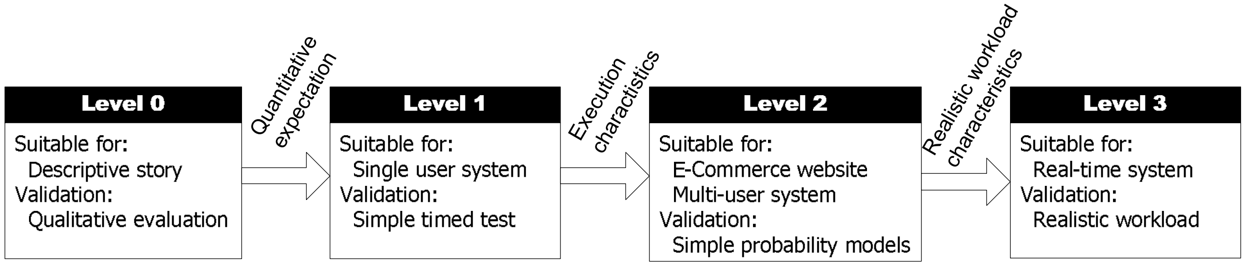

Figure 3. Performance requirements evolution model

variations. We call such details of NFRs system execution characteristics (SECs). SECs include information of what processes are in the system, and the frequency of the execution of the processes. A developer may identify SEC based on his or her experience. Software performance engineering [15, 16, 17] also provides a systematic way to identify SECs. A Level 1 PR graduates to Level 2 when SEC information is added.

Level 2 PRs are specified as quantitative performance objectives with the SECs of the system. An example of Level 2 PR is "*On average, twenty requests arrive at the server per minute. The authentication process accounts for 10% of incoming requests. The authentication process shall be completed in 0.2 seconds.*" To show that the performance objective is achieved, one needs to generate a workload that matches the SECs. A simple stochastic process, such as a Poisson process [14], can be used to represent the distribution of the workload.

3.1.4. Level 3. For real-time or performance-critical systems, the time constraints of an operation may have to be more strictly specified, or they may have to be met in worst-case conditions. For these systems, workload characteristics gathered from actual usage of the system can be more appropriate than probability models [2]. Level 3 represents the PRs with this kind of precision or worst-case specification. An example of a Level 3 worst-case PR is "*During the peak hours, 200 mobile shopping tablets are in use. 8% of the shoppers are either being served at the five checkout stations or are waiting in lines. For the rest of the customers, the promotional message shall display on the mobile tablet within 1 second after a customer enters a lane where the promotional items are located.*" The workload characteristics can be derived from the observation of user behaviors or from an older version of the system. Unless the project is performance-critical, Level 3 PRs are typically not needed.

3.2. PREM and agile requirements engineering

Auer and Beck suggest that performance criteria be established early and revised as the system matures [1]. PRs are handled in a similar fashion in PREM. Using PREM, PRs also start with a short description of performance. During the development, PRs specifications and corresponding test cases are refined to an appropriate level. PREM reminds the developers to find out more performance factors with performance engineering techniques and tools, which are indispensable for performance-critical systems.

A difference between a functional story and a performance specification is that the latter is not addible. At iteration planning, some stories are picked up to be implemented in the iteration. However, performance specifications cannot be implemented and added to the software system. When implementing a performance-critical story, the development team should write related performance test cases, based on the specification. The test cases are refined with more performance details discovered later in the software lifecycle. When the performance test case fails, the development team can discuss how and when to solve the problem.

3.3. Other considerations

A team might be tempted to skip levels and start from a high level PR specification. However, starting from a higher level is not an advised strategy. First of all, gathering information for a higher level PR takes time. For example, in one case study, twelve months were used to record the inputs to the system for the purpose of workload characterization [2]. Second, each level of PREM provides a new perspective of the system performance: Level 0 identifies which part of the system has performance requirements; Level 1 introduces a quantitative metric to measure the performance; Level 2 shows the interaction between a part and the overall system; Level 3 demonstrates how different components work together under realistic workloads

Specifying PRs at a lower level can sometimes suffice in practice even though the usage profile would indicate a need for additional specifications. For example, a short, rarely-used function might have little impact on the overall performance. Therefore a Level 1 specification will be enough for such functionality, even in a multiple-user system. We further observed that the presence of PRs and ongoing measurements within a test-driven development (TDD) [4] approach can give rise to a consistent progression of performance improvements that may obviate higher level PRs and associated testing. We shall discuss this experience in Section 4.

4. IBM experience

In this section, we use an IBM software project with a high focus on performance to show how the PRs are mapped to the PREM levels.

4.1. Performance requirements specification

An IBM software development group had been developing device drivers in C/C++ for over a decade. The group develops mission-critical software in the retail industry for its customers in a domain that demands high availability, correctness, reliability, and consistent and rapid throughput. Beginning in 2002, the group re-developed device drivers on a new platform in Java and adopted the TDD approach. Previous reports [12, 20] of this team have focused on their use of TDD to reduce post-release defects by 40%. Approximately 2390 automated JUnit test cases were written, including over 100 performance test cases.

Management was concerned about the potential performance implications of using Java for device drivers. As a result, performance was a focal point of the project throughout the development life cycle. The team was compelled to devise well-specified PRs as a means to demonstrate, in the early stages of the development, the feasibility of creating the device drivers in Java. The team utilized JUnit tests to monitor performance at the external wire, operating system kernel, and application levels, as the drivers were implemented.

Originally, rough performance requirements for the device drivers were collected from three primary sources. First, a small set of performance goals were distilled from direct customer feedback by representatives in marketing. These tended to be numerically inexact, i.e., Level 0 PRs, but they did serve to highlight specific areas where performance improvements beyond that of earlier versions were most desired.

Second, a domain expert specified which drivers were considered performance-critical versus non-performance-critical. Each performance-critical device driver had certain Level 1 performance metrics that were classified as requirements. For performance-critical drivers, the performance must exceed the drivers of the previous versions and be shown in quantitative metrics. Performance at least roughly comparable to that of the earlier versions was a goal for non-performance-critical drivers.

The third consideration was the set of limiting factors beyond which performance improvement was of little relevance. For example, the operating system kernel routines for port access, the internal USB hardware, and the external USB interface all contribute to the latency from command to physical device reaction. As long as the device operation was physically possible and acceptable within these constraints, these latencies were considered to form a bound on the performance of the device drivers.

4.2. Performance testing

The team designed performance tests that meet the high-level descriptions of the specification using the JUnit framework as code was being developed. This was done alongside the functional unit tests as an integral part of the TDD process. These performance test cases were designed to exercise the driver under test according to sample usage patterns specified by a domain expert.

In addition to individual test cases for different drivers, a “master” performance suite instantiated performance tests for every device. During development, unit tests were routinely run prior to check-in of new function, so the developers had early visibility to any severe problems caused by new code. Periodically, at snapshots called “levels,” the fully integrated system was subjected to the suite of functional tests and the master performance test suite. A record of performance test results was kept by level.

The development team did not expect that the performance tests results should improve monotonically with each new code level. In fact, as the code was made more robust by checking for and handling an increasing number of exception conditions, the opposite trend was sometimes expected. In practice, however, we observed that the performance of successive code levels generally did increase. We believe that this upward trend in performance was due in large part to the continual feedback provided by the built-in performance tests and the periodic tabulation of the full suite of performance test results. Contrary to a strategy of post-development “optimization” or “tuning”, the information provided by the performance test tools exercised as an integral part of development enabled a conscious tendency to design and code for better performance.

A final phase of performance testing for this project was a separate multi-threaded test case that activated multiple devices simultaneously (corresponding to Level 2 PRs). The degree of device activity overlap was varied while execution times were measured, in order to observe any potentially severe degradation induced by overlapping device driver activity. In practice, no severe degradations were observed over the spectrum of expected simultaneous device usage.

5. Conclusion and future work

Our primary contribution in this paper is the software performance requirements evolution model, PREM. Using the model as a guideline, a development team can identify and specify PRs incrementally, starting with casual descriptions, and refine them to a desired level of detail and precision. This evolutionary model fits in the "good enough" attitude of agile methods. Additionally, this model also helps the developers to write appropriate test cases for PRs. Furthermore, the IBM experience tells us PRs and performance test cases can be specified with appropriate level of detail, based on requirements. The experience also demonstrates TDD can be used to verify whether performance objectives have been met.

When PRs are evolved with more performance-related information, the corresponding test cases need to be modified to reflect the new performance characteristics. We are developing a testing framework to support evolutionary performance testing. The test framework will allow test writers to add more workload definition or to change a performance scenario with little modification. PREM along with this test framework will be the cornerstone for our further research about PRs specification and management.

6. Acknowledgements

The IBM device driver performance tests described in this paper were implemented by development team members Julio Sanchez, Raul Raudry, Susana Perez, Abraham Romo, Bobby Hargett, Dan Streetman, Kevin Bell, William White, and Phuc Do. This work is supported by The Center for Advanced Computing and Communication (CACC). We would like to thank the RealSearch reading group for their comments and help on the paper.

7. References

[1] Auer, K., and K. Beck, "Lazy Optimization: Patterns for Efficient Smalltalk Programming," in *Pattern Language of Program Design 2*, Addison-Wesley, Reading, MA, 1996. Also available at http://www.rolemodelsoftware.com/moreAboutUs/publications/articles/lazyopt.php.

[2] Avritzer, A., J. Kondek, D. Liu, and E. J. Weyuker, "Software Performance Testing Based on Workload Characterization," *Proceedings of the 3rd International Workshop on Software and Performance*, Rome, Italy, July, 2002, pp. 17-24.

[3] Beck, K., and M. Fowler, *Planning Extreme Programming*, Addison Wesley, Boston, MA, 2001.

[4] Beck, K., *Test Driven-Development by Example*, Addison Wesley, Boston, MA, 2003.

[5] Beck, K., *Extreme Programming Explained: Embrace Change, Second edition*, Addison Wesley, Boston, MA, 2005.

[6] Chung, L., B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[7] Clements, P. C., "Coming Attractions in Software Architecture," *Technical Report No. CMU/SEI-96-TR-008*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

[8] Clements, P. C. and L. M. Northrop, "Software Architecture: An Executive Overview," *Technical Report No. CMU/SEI-96-TR-003*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

[9] Crispin, L., and T. House, *Testing Extreme Programming*, Addison-Wesley, Boston, MA, 2003.

[10] Fox, G., "Performance Engineering as a Part of the Development Life Cycle for Large-Scale Software Systems," *Proceedings of the 11th International Conference of Software Engineering*, Nice, France, March, 1990, pp. 52-62.

[11] Gao, J. Z., H.-S. J. Tsao, and Y. Wu, *Testing and Quality Assurance for Component-Based Software*, Artech House, Norwood, MA, 2003.

[12] Maximilien, E. M. and L. Williams, "Assessing Test-Driven Development at IBM," *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003, pp. 564-569.

[13] Paetsch, F., A. Eberlein, and F. Maurer, "Requirements Engineering and Agile Software Development," *Proceedings of the 12th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Linz, Austria, June 2003, pp. 308-313.

[14] Ross, S. M., *Introduction to Probability Models, 8th Edition*, Academic Press, Burlington, MA, 2003.

[15] Smith, C. U., *Performance Engineering of Software Systems*, Addison-Wesley, Reading, MA, 1990.

[16] Smith, C. U. and L. G. Williams, "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives," *IEEE Transaction on Software Engineering*, vol. 19, no. 7, July 1993, pp. 720-741.

[17] Smith C. U. and L. G. Williams, Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley, Boston, MA, 2002.

[18] Sommerville, I., *Software Engineering 7th Edition*, Addison-Wesley, Boston, MA, 2004.

[19] Weyuker, E. J., and F. I. Vokolos, "Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study," *IEEE Transactions on Software Engineering*, vol. 26, no. 12, December 2000, pp. 1147-1156.

[20] Williams, L., E. M. Maximilien, and M. Vouk, "Test-Driven Development as a Defect Reduction Practice," *Proceedings of the 14th International Symposium on Software Reliability Engineering*, Denver, CO, November, 2003, pp. 34-35.

[21] Zadrozny, P., *J2EE Performance Testing with BEA WebLogic Server*, Expert Press, Birmingham, UK, 2002.