Agile Record

The Magazine for Agile Developers and Agile Testers

Requirements in Agile Projects

www.agilerecord.com free digital version made in Germany ISSN 2191-1320

August 2012

issue 11

Performance Requirements in Agile Projects

by Alexander Podelko

What is Special?

It looks like agile methodologies are somewhat struggling with performance requirements (and non-functional requirements in general). Probably there are several reasons for that. One is that actually even traditional software development methodologies and processes never came with a good approach to handle performance requirements. They are, of course, considered in both literature and practical projects – but are usually handled rather in ad hoc manner. Actually the process of gathering and elaboration of performance requirements is rather agile in itself, and attempts to make it rigorous and formal look unnatural and have never fully succeeded – so it should be easier and more natural to do it as part of agile methods. Still the challenge of handling multi-dimensional and difficult to formalize performance requirements remains intact and the difference is rather in minor adjustments1 to agile processes than in the essence of performance requirements.

Another reason is that practical agile development is struggling with performance in general. Theoretically it should be a piece of cake: every iteration you have a working system and know exactly where you stand with the system's performance. You shouldn't wait until the end of the waterfall process to figure out where you are – on every iteration you can track your performance against requirements and see the progress (making adjustments on what is already implemented and what is not yet). Clearly it is supposed to make the whole performance engineering process much more straightforward.

Unfortunately, it looks like it doesn't always work this way in practice. So such notions as "hardening iterations" and "technical debt" get introduced. Although it is probably the same old problem: functionality gets priority over performance (which is somewhat explainable: you need first some functionality before you can talk about its performance). So performance related activities slip toward the end of the project, and the chance is missed to implement a proper performance engineering process built around performance requirements. Another issue here is that agile methods are oriented toward breaking projects into small tasks, which is quite difficult to do with performance (and many other non-functional requirements)6 – performance-related activities usually span the whole project.

Let's now consider performance requirements in detail, keeping these issues in mind, to see if the statements above have some ground.

Performance metrics

Before diving into details of the performance requirements process, let's discuss the most important performance metrics (sometimes referred to as Key Performance Indicators, KPIs). It is a challenge to get all stakeholders to agree on specific metrics and ensure that they can be measured in a compatible way at every stage of the lifecycle (which may require specific monitoring tools and application instrumentation).

Let's take a high-level view of a system (Fig.1). On one side we have users who use the system to satisfy their needs. On another side we have the system, a combination of hardware and software, created (or to be created) to satisfy users' needs.

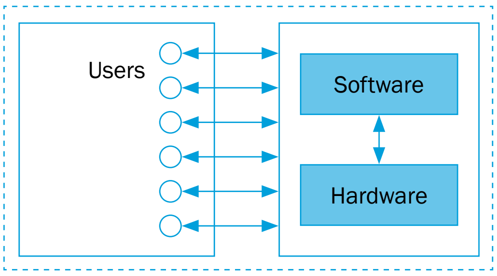

Fig.1. A high-level view of a system.

Business performance requirements

Users are not interested in what is inside the system and how it functions as soon as their requests get processed in a timely manner (leaving aside personal curiosity and subjective opinions). So business requirements should state how many requests of each kind must go through the system (throughput) and how quickly they need to be processed (response times). Both parts are vital: good throughput with long response times usually is as unacceptable as are good response times with low throughput. Throughput is a business requirement, whereas response times have both usability and business components.

Throughput is the rate at which incoming requests are completed. Throughput defines the load on the system and is measured in operations per time period. It may be the number of transactions per second or the number of processed orders per hour. In most cases we are interested in a steady mode when the number of incoming requests would be equal to the number of processed requests.

Defining throughput may be pretty straightforward for a system doing the same type of business operations all the time, like processing orders or printing reports when they are homogenous. Clustering requests into a few groups, such as small, medium, and large reports, may be needed if requests differ significantly. It may be more difficult for systems with complex workloads because the ratio of different types of requests can change with the time and season.

Homogenous throughput with randomly arriving requests (sometimes assumed in modeling and requirements analysis) is a simplification in most cases. Throughput usually varies with time. For example, throughput can be defined for a typical hour, peak hour, and non-peak hour for each particular kind of load. In environments with fixed hardware configuration the system should be able to handle peak load, but in virtualized or cloud environments it may be helpful to further detail what the load is hour-by-hour to ensure better hardware utilization.

Quite often, however, the load on the system is characterized by the number of users. Partially it comes from the business (in many cases the number of users is easier to find out), and partially it comes from performance testing: Unfortunately, quite often performance requirements get defined during performance testing and the number of users is the main lever to manage load in load generation tools.

However, the number of users doesn't, by itself, define throughput. Without defining what each user is doing and how intensely (i.e. throughput for one user), the number of users doesn't make much sense as a measure of load. For example, if 500 users are each running one short query each minute, we have throughput of 30,000 queries per hour. If the same 500 users are running the same queries, but only one query per hour, the throughput is 500 queries per hour. So there may be the same 500 users, but a 60X difference between loads (and at least the same difference in hardware requirements for the application – probably more, considering that not all systems achieve linear scalability).

In addition to different kinds of requests, most systems use sessions: some system resources are associated with the user (source of requests). So the number of parallel users (sessions) would be an important requirement further qualifying throughput. In a more generic way this metric may be named concurrency: the number of simultaneous users or threads. It is important: even inactive, but connected users still hold some resources. The number of online users (the number of parallel sessions) looks like the best metric for concurrency (complementing throughput and response time requirements). However, terminology is somewhat vague here, sometimes "the number of users" has a different meaning. For example, it may be named or "truly concurrent" users.

Response times (in the case of interactive work) or processing times (in the case of batch jobs or scheduled activities) define how fast requests should be processed. Acceptable response times should be defined in each particular case. A time of 30 minutes could be excellent for a big batch job, but absolutely unacceptable for accessing a web page in a customer portal. Response times depend on workload, so it is necessary to define conditions under which specific response times should be achieved; for example, a single user, average load or peak load.

Response time is the time in the system (the sum of queuing and processing time). Usually there is always some queuing time because the server is a complex object with sophisticated collaboration of multiple components including processor, memory, disk system, and other connecting parts. That means that response time is larger than service time (to use in modeling) in most cases.

Significant research has been done to define what the response time should be for interactive systems, mainly from two points of view: what response time is necessary to achieve optimal user's performance (for tasks like entering orders), and what response time is necessary to avoid web site abandonment (for the Internet). Most researchers agreed that for most interactive applications there is no point in making the response time faster than one to two seconds, and it is helpful to provide an indicator (like a progress bar) if it takes more than eight to 10 seconds.

Response times for each individual transaction vary, so we need to use some aggregate values when specifying performance requirements, such as averages or percentiles (for example, 90 percent of response times are less than X). The Apdex standard18 uses a single number to measure user satisfaction.

It is very difficult to consider performance (and, therefore, performance requirements) without full context. It depends, for example, on the volume of data involved, hardware resources provided, and functionality included in the system. So if any of that information is known, it should be specified in the requirements. Not everything may be specified at the same point: while the volume of data is usually determined by the business and should be documented at the beginning, the hardware configuration is usually determined during the design stage.

Technological performance requirements

The performance metrics of the system (the right side of the fig.1) are not important from the business (or user) point of view, but are very important for IT (people who create and operate the system). These internal (technological) requirements are derived from business and usability requirements during design and development and are very important for the later stages of the system lifecycle. Traditionally such metrics were mainly used for monitoring and capacity management because they are easier to measure, and only recently tools measuring end-user performance get some traction.

The most wide-spread metric, especially in capacity management and production monitoring, is resource utilization. The main groups of resources are CPU, I/O, memory, and network. However, the available hardware resources are usually a variable in the beginning. It is one of the goals of the design process to specify hardware needed for the system from the business requirements and other inputs like company policies, available expertise, and required interfaces.

When resource requirements are measured as resource utilization, they are related to a particular hardware configuration. They are meaningful metrics when the hardware configuration is known. But these metrics don't make sense as requirements until the hardware configuration would be decided upon; how can we talk, for example, about processor utilization if we don't know yet how many processors we would have? And such requirements are not useful as requirements for software if it gets deployed to different hardware configurations, and, especially, for Commercial Off-the-Shelf (COTS) software.

The only way we can speak about resource utilization in early phases of the system lifecycle is as a generic policy. For example, corporate policy may be that CPU utilization should be below 70 percent.

When required resources are specified in absolute values, like the number of instructions to execute or the number of I/O operations per transaction (as sometimes used, for example, for modeling), it may be considered as a performance metric of the software itself, without binding it to a particular hardware configuration. In the mainframe world, MIPS was often used as such a metric for CPU consumption, but there is no such widely used metric in the distributed systems world.

The importance of resource-related requirements is increasing again with the trends of virtualization, cloud computing, and service-oriented architectures. When we depart from the "server(s) per application" model, it becomes difficult to specify requirements as resource utilization, as each application will add only incrementally to resource utilization. There are attempts to introduce such metrics. For example, the 'CPU usage in MHz' or 'usagemhz' metric used in the VMware world, or the 'Megacycles' metric sometimes used by Microsoft14. Another related metric sometimes (but rarely) used is efficiency when it is defined as throughput divided by resources (however, the term is often used differently).

In the ideal case (for example, when the system is CPU bound and we can scale the system linearly by just adding processors), we can easily find the needed hardware configuration if we have an absolute metric of resources required.

For example, if software needs X units of hardware power per request and a processor has Y units of hardware power, we can calculate the number of such processors N needed for processing Z requests as N=Z*X/Y. The reality, of course, is more sophisticated. First of all, we have different kinds of hardware resources: processors, memory, I/O, and network. Usually we concentrate on the most critical one keeping in mind others as restrictions.

Scalability is a system's ability to meet the performance requirements as the demand increases (usually by adding hardware). Scalability requirements may include demand projections such as increases in the number of users, transaction volumes, data sizes, or adding new workloads. How response times will increase with increasing load or data is important too (load or data sensitivity).

From a performance requirements perspective, scalability means that you should specify performance requirements not only for one configuration point, but as a function of load or data. For example, the requirement may be to support throughput increase from five to 10 transactions per second over the next two years with a response time degradation of not more than 10 percent.

Scalability is also a technological (internal IT) requirement, or perhaps even a "best practice" of systems design. From the business point of view, it is not important how the system is maintained to support growing demand. If we have growth projections, we probably need to keep the future load in mind during the system design and have a plan for adding hardware as needed.

Requirements process

The IEEE Software Engineering Book of Knowledge11 defines four stages of requirements process:

- Elicitation: Identifying sources and collecting requirements.
- Analysis: Classifying, elaborating, and negotiating requirements.
- Specification: Producing a document. While documenting requirements is important, the way to do this depends on the software development methodology used, corporate standards, and other factors.
- Validation: Making sure that requirements are correct.

Seeing the words 'elaborating' and 'negotiating' in the stage descriptions, we may assume that it should fit agile methods well. Elicitation would match initial requirements gathering (such a creation of user stories) and Analysis – Specification – Validation fit well in the iterative process when in each iteration we elaborate these requirements further in close cooperation with all stakeholders.

Let's consider each stage and its connection with other software life cycle processes.

Elicitation

We may classify performance requirements into business, usability, and technological requirements.

Business requirements come directly from the business and may be captured very early in the project lifecycle, before design starts. For example, a customer representative should enter 20 requests per hour and the system should support up to 1,000 customer representatives. Translated into more technical terms, the requests should be processed in five minutes on average, throughput would be up to 20,000 requests per hour, and there could be up to 1,000 parallel user sessions.

The main trap here is to immediately link business requirements to a specific design, technology, or usability requirements, thus limiting the number of available design choices. If we consider a web system, for example, it is probably possible to squeeze all the information into a single page or have a sequence of two dozen screens. All information can be saved at once at the end, or each page of these two dozen pages can be saved separately. We have the same business requirements, but response times per page and the number of pages per hour would be different.

While the final requirements should be quantitative and measurable, it is not an absolute requirement for initial requirements. Scott Barber, for example, advocates that we need to gather qualitative requirements first3. While business people know what the system should do and may provide some numeric information, they are usually not trained in requirement elicitation and system design. If asked to provide quantitative and measurable requirements, they may finally provide them based on whatever assumptions they have about system's design and human-computer interaction, but quite often it results in wrong assumptions being documented as business requirements. We should document real business needs in the form in which they are available (perhaps as user stories from the business point of view), and only then elaborate them into quantitative and measurable requirements (during the project's iterations).

One often missed issue, as Scott Barber notes, is goals versus requirements3. Most of response time "requirements" (and sometimes other kinds of performance requirements) are goals, not requirements. They are something that we want to achieve, but missing them won't necessarily prevent deploying the system.

In many cases, especially for response times, there is a big difference between goals and requirements (the point when stakeholders agree that the system can't go into production with such performance). For many corporate web applications, response time goals are two to five seconds, and requirements may be somewhere between eight seconds and a minute.

Determining what the specific performance requirements are is another large topic that is difficult to formalize. Consider the approach suggested by Peter Sevcik for finding T, the threshold between satisfied and tolerating users. T is the main parameter of the Apdex (Application Performance Index) methodology, providing a single metric of user satisfaction with the performance of enterprise applications. Peter Sevcik defined ten different methods18.

- Default value (the Apdex methodology suggests 4 sec)
- Empirical data
- User behavior model (number of elements viewed / task repetitiveness)
- Outside references
- Observing the user
- Controlled performance experiment
- Best time multiple
- Find frustration threshold F first, and calculate T from F (the Apdex methodology assumes that F = 4T)
- Interview stakeholders
- Mathematical inflection point

The idea is the use of several of these methods for the same system. If all come to approximately the same number, they give us T. While this approach was developed for production monitoring, there is definitely a strong correlation between T and the response time goal (having all users satisfied sounds like as a pretty good goal), and between F and the response time requirement. So the approach probably can be used for getting response time requirements with minimal modifications. While some specific assumptions like four seconds for default or the F=4T relationship may be up for argument, the approach itself conveys the important message that there are many ways to determine a specific performance requirement, and it would be better for validation purposes to get it from several sources. Depending on your system, you can determine which methods from the above list are applicable (or what other methods may make sense in your particular case), get the metrics and determine your requirements.

Usability requirements, mainly related to response times, are based on the basic principles of human-computer interaction. Many researchers agree that users lose focus if response times are more than 8 to 10 seconds and that making the response time faster than one to two seconds doesn't help productivity much. These usability considerations may influence design choices (such as using several web pages instead of one). In some cases, usability requirements are linked closely to business requirements; for example, make sure that your system's response times are not worse than the response times of similar or of competitor's systems.

As long ago as 1968, Robert Miller's paper 'Response Time in Man-Computer Conversational Transactions' described three threshold levels of human attention15. Jakob Nielsen believes that Miller's guidelines are fundamental for human-computer interaction, so they are still valid and not likely to change with whatever technology comes next16. These three thresholds are:

- Users view response time as instantaneous (0.1-0.2 second)
- Users feel they are interacting freely with the information (1-5 seconds)
- Users are focused on the dialog (5-10 seconds)

Users view response time as instantaneous (0.1-0.2 second): Users feel that they directly manipulate objects in the user interface. For example, the time from the moment the user selects a column in a table until that column highlights or the time between typing a symbol and its appearance on the screen. Robert Miller reported that threshold to be 0.1 seconds. According to Peter Bickford15 0.2

second forms the mental boundary between events that seem to happen together and those that appear as echoes of each other5.

Although it is a quite important threshold, it is often beyond the reach of application developers. That kind of interaction is provided by operating system, browser, or interface libraries, and usually happens on the client side, without interaction with servers (except for dumb terminals, that is rather an exception for business systems today).

Users feel they are interacting freely with the information (1-5 seconds): They notice the delay, but feel the computer is "working" on the command. The user's flow of thought stays uninterrupted. Robert Miller reported this threshold as one-two seconds15.

Peter Sevcik identified two key factors impacting this threshold17: the number of elements viewed and the repetitiveness of the task. The number of elements viewed is, for example, the number of items, fields, or paragraphs the user looks at. The amount of time the user is willing to wait appears to be a function of the perceived complexity of the request. The complexity of the user interface and the number of elements on the screen both impact the thresholds. Back in the 1960s through 1980s, the terminal interface was rather simple and a typical task was data entry, often one element at a time. So earlier researchers reported that one to two seconds was the threshold to keep maximal productivity. Modern complex user interfaces with many elements may have higher response times without adversely impacting user productivity. Users also interact with applications at a certain pace depending on how repetitive each task is. Some are highly repetitive; others require the user to think and make choices before proceeding to the next screen. The more repetitive the task is, the better the expected response time.

That is the threshold that gives us response time usability goals for most user-interactive applications. Response times above this threshold degrade productivity. Exact numbers depend on many difficult-to-formalize factors, such as the number and types of elements viewed or repetitiveness of the task, but a goal of two to five seconds is reasonable for most typical business applications.

There are researchers who suggest that response time expectations increase with time. Forrester research8 suggests two seconds response time; in 2006 similar research suggested four seconds (both research efforts were sponsored by Akamai, a provider of web accelerating solutions). While the trend probably exists, the approach of this research was often questioned because they just asked users. It is known that user perception of time may be misleading. Also, as mentioned earlier, response time expectations depends on the number of elements viewed, the repetitiveness of the task, user assumptions of what the system is doing, and UI showing the status. Stating a standard without specification of what page we are talking about may be overgeneralization.

Users are focused on the dialog (5-10 seconds): They keep their attention on the task. Robert Miller reported that threshold as 10 seconds15. Users will probably need to reorient themselves when they return to the task after a delay above this threshold, so productivity suffers.

Peter Bickford investigated user reactions when, after 27 almost instantaneous responses, there was a two-minute wait loop for the 28th time for the same operation. It took only 8.5 seconds for half the subjects to either walk out or hit the reboot5. Switching to a watch cursor during the wait delayed the subject's departure for about 20 seconds. An animated watch cursor was good for more than a minute, and a progress bar kept users waiting until the end. Bickford's results were widely used for setting response times requirements for web applications.

That is the threshold that gives us response time usability requirements for most user-interactive applications. Response times above this threshold cause users to lose focus and lead to frustration. Exact numbers vary significantly depending on the interface used, but it looks like response times should not be more than 8 to 10 seconds in most cases. Still, the threshold shouldn't be applied blindly; in many cases, significantly higher response times may be acceptable when an appropriate user interface is implemented to alleviate the problem.

Analysis and specification

The third category, technological requirements, comes from chosen design and used technology. Some technological requirements may be known from the beginning if some design elements are given, but others are derived from business and usability requirements throughout the design process and depend on the chosen design.

For example, if we need to call ten web services sequentially to show the web page with a three-second response time, the sum of response times of each web service, the time to create the web page, transfer it through the network and render it in a browser should be below 3 seconds. That may be translated into response time requirements of 200-250 milliseconds for each web service. The more we know, the more accurately we can apportion overall response time to web services.

Another example of technological requirements is resource consumption requirements. For example, CPU and memory utilization should be below 70% for the chosen hardware configuration.

Business requirements should be elaborated during iterations and merge together with usability and technological requirements into the final performance requirements, which can be verified during testing and monitored in production. The main reason why we separate these categories is to understand where the requirement comes from. Is it a fundamental business requirement and the system fails if we miss it, or is it a result of a design decision that may be changed if necessary.

A significant difference between traditional and agile methods is in specification. Traditional requirements engineering / architect's vocabulary is very different from the terminology used in development, performance testing, or capacity planning. Performance and scalability are often referred to as examples of Quality Attributes (QA), a part of Non-functional Requirements (NFR).

In addition to specifying requirements in plain text, there are multiple approaches to formalize documenting of requirements. For example, Quality Attribute Scenarios by The Carnegie Mellon Software Engineering Institute (SEI) or Planguage (Planning Language) introduced by Tom Gilb.

The QA scenario defines source, stimulus, environment, artifact, response, and response measure4. For example, the scenario may be that users initiate 1,000 transactions per minute randomly under normal operations, and these transactions are processed with an average latency of two seconds. For this example:

- Source is a collection of users.
- Stimulus is the random initiation of 1,000 transactions per minute.
- Artifact is always the system's services.
- Environment is the system state, normal mode in our example.
- Response is processing the transactions.
- Response measure is the time it takes to process the arriving events (an average latency of two seconds in our example).

Planguage was suggested by Tom Gilb and may work better for quantifying quality requirements19. Planguage keywords include:

- Tag: a unique identifier
- Gist: a short description
- Stakeholder: a party materially affected by the requirement
- Scale: the scale of measure used to quantify the statement
- Meter: the process or device used to establish location on a Scale
- Must: the minimum level required to avoid failure
- Plan: the level at which good success can be claimed
- Stretch: a stretch goal if everything goes perfectly
- Wish: a desirable level of achievement that may not be attainable through available means
- Past: an expression of previous results for comparison
- Trend: an historical range or extrapolation of data
- Record: the best known achievement

It is very interesting that Planguage defines four levels for each requirement: minimum, plan, stretch, and wish.

There is no standard approach to specifying performance requirements in agile methods. Mostly it is suggested to present them as user stories7,10 or as constraints13. And the difference is not so much in the way the requirements are presented, both ways rather use plain text. User stories assume using a user voice form. Cohn, for example, suggests to use the "As a <type of user>, I want <some goal>, so that <some reason>" template7 for user stories (although he cautions that the user story template should only be used as a thinking tool, it should not be used as a fixed template). For constraints, both traditional expressions and user voice forms may be used13.

The difference between user stories and constraints approaches is not in performance requirements per se, but how to address them during the development process. The point of the constraint approach is that user stories should represent finite manageable tasks, while performance-related activities can't be handled as such because they usually span multiple components and iterations. Those who suggest to use user stories address that concern in another way – for example, separating cost of initial compliance and cost of ongoing compliance9.

Another question is how to specify response time requirements or goals. Individual transaction response times vary, so aggregate values should be used. For example, such metrics as average, maximum, different kinds of percentiles, or median. The problem is that whatever aggregate value you use, you lose some information.

Percentiles are more typical in SLAs (Service Level Agreements). For example, 99.5 percent of all transactions should have a response time of less than five seconds. While that may be sufficient for most systems, it doesn't answer all questions. What happens with the remaining 0.5 percent? Do these 0.5 percent of transactions finish in six to seven seconds, or do all of them timeout? You may need to specify a combination of requirements: for example, average four seconds and maximal 12 seconds, or average four seconds and 99 percent below 10 seconds.

Moreover, there are different viewpoints for performance data that need to be provided for different audiences. You need different metrics for management, engineering, operations, and quality assurance. For operations and management, percentiles may work best. If you do performance tuning and want to compare two different runs, average may be a better metric to see the trend. For design and development, you may need to provide more detailed metrics; for example, if the order processing time depends on the number of items in the order, it may be separate response time metrics for one to two, three to 10, 10 to 50, and more than 50 items.

Often different tools are used to provide performance information to different audiences; they present information in a different way and may measure different metrics. For example, load testing tools and active monitoring tools provide metrics for the used synthetic workload that may differ significantly from the actual production load. This becomes a real issue if you want to implement some kind of process, such as ITIL Continual Service Improvement or Six Sigma, to keep performance under control throughout the whole system lifecycle.

Things get more complicated when there are many different types of transactions, but a combination of percentile-based performance and availability metrics usually works in production for most interactive systems. While more sophisticated metrics may be necessary for some systems, in most cases they make the process overcomplicated and results difficult to analyze.

There are efforts to make an objective user satisfaction metric. For example, Apdex, Application Performance Index18, is a single metric of user satisfaction with the performance of enterprise applications. The Apdex metric is a number between 0 and 1, where 0 means that no users were satisfied, and 1 means all users were satisfied. The approach introduces three groups of users: satisfied, tolerating, and frustrated. Two major parameters are introduced: threshold response times between satisfied and tolerating users T, and between tolerating and frustrated users F. There probably is a relationship between T and the response time goal, and between F and the response time requirement. However, while Apdex may be a good metric for management and operations, it is probably too high-level for engineering.

Validation and verification

Requirements validation is making sure that requirements are valid (although the term 'validation' is quite often used in the meaning of checking against test results instead of verification). A good way to validate a requirement is to get it from different independent sources: if all numbers are about the same, it is a good indication that the requirement is probably valid. Validation may include, for example, reviews, modeling, and prototyping. Requirements process is iterative by nature and requirements may change with time, so to be able to validate them it is important to trace requirements back to their source.

Requirements verification is checking if the system performs according to the requirements. To make meaningful comparisons, both the requirements and results should use the same metrics. One consideration here is that many load testing and many monitoring tools measure only server and network time. While end user response times, which business is interested in and which is usually assumed in performance requirements, may differ significantly, especially for rich web clients or thick clients due to client-side processing and browser rendering. Verification should be done using load testing results as well as during ongoing production monitoring. Checking production monitoring results against requirements and load testing results is also a way to validate that load testing was done properly.

Requirement verification presents another subtle issue: how to differentiate performance issues from functional bugs exposed under load. Often, additional investigation is required before you can determine the cause of your observed results. Small anomalies from expected behavior are often signs of bigger problems, and you should at least figure out why you get them.

When 99 percent of your response times are three to five seconds (with the requirement of five seconds) and 1 percent of your response times are five to eight seconds, it usually is not a problem. However, it probably should be investigated if this 1 percent fail or have strangely high response times (for example, more than 30 sec) in an unrestricted, isolated test environment. This is not due to some kind of artificial requirement, but is an indication of an anomaly in system behavior or test configuration. This situation often is analyzed from a requirements point of view, but it shouldn't be, at least not until the reasons for that behavior become clear.

These two situations look similar, but are completely different in nature:

1. The system is missing a requirement, but results are consistent. This is a business decision, such as a cost vs. response time trade-off.
2. Results are not consistent (while requirements can even be met). That may indicate a problem, but its scale isn't clear until investigated.

Unfortunately, this view is rarely shared by development teams too eager to finish the project, move it into production, and move on to the next project. Most developers are not very excited by the prospect of debugging code for small memory leaks or hunting for a rare error that is difficult to reproduce. So the development team becomes very creative in finding "explanations". For example, growing memory and periodic long-running transactions in Java are often explained as a garbage collection issue. That is false in most cases. Even in the few cases when it is true, it makes sense to tune garbage collection and prove that the problem went away.

Another typical situation is getting some transactions failed during performance testing. It may still satisfy performance requirements, which, for example, state that 99% of transactions should be below X seconds – and the share of failed transaction is less than 1 percent. While this requirement definitely makes sense in production, where we may have network and hardware failures, it is not clear why we get failed transactions during the performance test if it was run in a controlled environment and no system failures were observed. It may be a bug exposed under load or a functional problem for some combination of data.

When some transactions fail under load or have very long response times in the controlled environment and we don't know why, we have one or more problems. When we have unknown problems, why not trace them down and fix them in the controlled environment? It would be much more difficult in production. What if these few failed transactions are a view page for your largest customer and you won't be able to create any order for this customer until the problem is fixed? In functional testing, as soon as you find a problem, you usually can figure out how serious it is. This is not the case for performance testing: usually you have no idea what caused the observed symptoms and how serious it is, and quite often the original explanations turn out to be wrong.

As Richard Feynman said in his appendix to the Rogers Commission Report on the Challenger space shuttle accident12, "The equipment is not operating as expected, and therefore there is a danger that it can operate with even wider deviations in this unexpected and not thoroughly understood way. The fact that this danger did not lead to a catastrophe before is no guarantee that it will not the next time, unless it is completely understood."

Summary

We need to specify performance requirements at the beginning of any project for design and development (and, of course, reuse them during performance testing and production monitoring). While performance requirements are often not perfect, forcing stakeholders just to think about performance increases the chances of project success. Agile methods provide a unique opportunity to verify performance requirements early and track performance through all iterations.

What exactly should be specified – goal vs. requirements (or both), average vs. percentile vs. APDEX, etc. – depends on the system and environment. Whatever it is, it should be elaborated into quantitative and measurable in the end. Making requirements too complicated may hurt. We need to find meaningful goals and requirements, not invent something just to satisfy some bureaucratic process.

If we define performance requirements in the beginning of the project, they become the backbone of the performance engineering process and we can use and elaborate them throughout all iterations and track our progress from the performance engineering point of view. Continuing to trace them in production creates a performance feedback loop providing us with input to system maintenance and future development.

References

1. Agile Non-Functional Requirements. 2009.
   http://tynerblain.com/blog/2009/02/10/agile-non-functional-reqs/
2. Ambler, S.W. Beyond Functional Requirements On Agile Projects. Dr.Dobb's, 2008.
   http://www.drdobbs.com/architecture-and-design/210601918
3. Barber, S. Get performance requirements right – think like a user, Compuware, 2007.
   http://www.perftestplus.com/resources/requirements_with_compuware.pdf
4. Bass L., Clements P., Kazman R. Software Architecture in Practice, Addison-Wesley, 2003.
   http://etutorials.org/Programming/Software+architecture+in+practice,+second+edition
5. Bickford P. Worth the Wait? Human Interface Online, View Source, 10/1997.
   http://web.archive.org/web/20040913083444/http://developer.netscape.com/viewsource/bickford_wait.htm
6. Cohn, M. Estimating Non-Functional Requirements. 2011.
   http://www.mountaingoatsoftware.com/blog/estimating-non-functional-requirements
7. Cohn, M. Non-functional Requirements as User Stories. 2008.
   http://www.mountaingoatsoftware.com/blog/non-functional-requirements-as-user-stories/
8. eCommerce Web Site Performance Today. Forrester Consulting on behalf of Akamai Technologies, 2009.
   http://www.akamai.com/html/about/press/releases/2009/press_091409.html
9. Hazrati, V. Nailing Down Non-Functional Requirements. InfoQ, 2011.
   http://www.infoq.com/news/2011/06/nailing-quality-requirements
10. Howard, K. Handling Non-Functional Requirements On an Agile Project. Agile 2009.
    http://www.slideshare.net/kenhoward01/handling-non-functional-requirements-on-an-agile-project
11. Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE, 2004.
    http://www.computer.org/portal/web/swebok
12. Feynman R.P. Appendix F – Personal observations on the reliability of the Shuttle.
    http://science.ksc.nasa.gov/shuttle/missions/51-l/docs/rogers-commission/Appendix-F.txt
13. Leffingwell D, Shriver R.Nonfunctional Requirements (System Qualities) Agile Style. Agile 2010.
    http://www.theagileengineer.com/public/Home/Entries/2010/8/12_Agile_2010_Presentation__Non_Functional_Requirements_(Qualities),_Agile_Style.html
14. Mailbox Server Processor Capacity Planning.
    http://technet.microsoft.com/en-us/library/ee712771.aspx
15. Miller, R. B. Response time in user-system conversational transactions, In Proceedings of the AFIPS Fall Joint Computer Conference, 33, 1968, 267-277.
16. Nielsen J. Response Times: The Three Important Limits, Excerpt from Chapter 5 of Usability Engineering, 1994.
    http://www.useit.com/papers/responsetime.html
17. Sevcik, P. How Fast Is Fast Enough, Business Communications Review, March 2003.
    http://www.bcr.com/architecture/network_forecasts%10sevcik/how_fast_is_fast_enough?_20030315225.htm
18. Sevcik, P. Using Apdex to Manage Performance, CMG, 2008.
    http://www.apdex.org/documents/Session318.0Sevcik.pdf
19. Simmons E. Quantifying Quality Requirements Using Planguage, Quality Week, 2001.
    http://www.clearspecs.com/downloads/ClearSpecs20V01_Quantifying%20Quality%20Requirements.pdf

> About the author

Alex Podelko

For the last fifteen years Alex Podelko has worked as a performance engineer and architect for several companies. Currently he is Consulting Member of Technical Staff at Oracle, responsible for performance testing and optimization of Hyperion products. Alex serves as a director for the Computer Measurement Group (CMG) http://cmg.org, a volunteer organization of performance and capacity planning professionals. He blogs at http://alexanderpodelko.com/blog and can be found on Twitter as @apodelko.