Part II

PERFORMANCE REQUIREMENTS:

An Attempt at a Systematic View

In the May/June 2011 issue of ST&QA Magazine, in the first part of the article, we discussed the most important performance metrics. Now we will discuss all stages of the performance requirements process, which include elicitation, analysis, specification, and validation, according to the IEEE Software Engineering Book of Knowledge (SWEBOK).

by Alex**PODELKO**

IEEE Software Engineering Book of Knowledge defines four stages or requirements[1]:

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

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

Elicitation

If we look at the performance requirements from another point of view, we can classify them 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 1000 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 in the end or each page of these two-dozen 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 first[2]. 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 the system's design and human-computer interaction, but quite often it results in wrong assumptions being documented as business requirements. We need to document real business requirements in the form they are available, and only then elaborate them into quantitative and measurable efforts.

One often missed issue, as Scott Barber notes, is goals versus requirements[2]. 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 interactive web applications, response time goals are two to five seconds and requirements may be somewhere between eight seconds and a minute.

One approach may be to define both goals and requirements. The problem is that, except when coming from legal or contractual obligation, requirements are very difficult to get. Even if stakeholders define performance requirements, quite often, when it comes to the go/no go decision, it becomes clear that it was not the real requirements, but rather second-tier goals.

In addition, multiple performance metrics only together provide the full picture. For example, you may have a 10-second requirement and you get 15-second response time under the full load. But what if you know that this full load is the high load on the busiest day of year, that response times for the maximal load for other days are below 10 seconds, and you see that it is CPU-constrained and may be fixed by a hardware upgrade? Real response time requirements are so environment and business dependent that for many applications it may be problematic to force people to make hard decisions in advance for each possible combination of circumstances. One approach may be to specify goals (making sure that they make sense) and only then, if they are not met, make the decision what to do with all the information available.

Determining what 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 methods [3]

1. Default value (the Apdex methodology suggest 4 sec)
2. Empirical data
3. User behavior model (number of elements viewed / task repetitiveness)
4. Outside references
5. Observing the user
6. Controlled performance experiment
7. Best time multiple
8. Find frustration threshold F first and calculate T from F (the Apdex methodology assumes that F = 4T)
9. Interview stakeholders
10. Mathematical inflection point

Each method is discussed in detail in *Using Apdex to Manage Performance.*

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 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 to get it from several sources for validation purposes. Depending on your system, you can determine which methods from the above list (or maybe some others) are applicable, calculate the metrics and determine your requirements.

> "Usability requirements, mainly related to response times, are based on the basic principles of human-computer interaction."
>
> Alex PODELKO

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 response times should be 2 to 5 seconds for maximum productivity. 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 response times of similar or competitor systems.

As long ago as 1968, Robert Miller's paper *Response Time in Man-Computer Conversational Transactions* described three threshold levels of human attention[4]. 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 next[5]. These three thresholds are:

1. Users view response time as instantaneous (0.1-0.2 second)
2. Users feel they are interacting freely with the information (1-5 seconds)
3. 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 as 0.1 seconds. According to Peter Bickford 0.2 second forms the mental boundary between events that seem to happen together and those that appear as echoes of each other[6].

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 seconds[4].

Peter Sevcik identified two key factors impacting this threshold[7]: 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. Impacting thresholds are the complexity of the user interface and the number of elements on the screen. Back in 1960s through 1980s the terminal interface was rather simple and a typical task was data entry, often one element at a time. 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 research of 2009 suggests two second response time; in 2006 similar research suggested four seconds (both research efforts were sponsored by Akamai, a provider of web accelerating solutions).[8] 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 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 threshold as 10 seconds[4]. 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 reboot[6]. 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 eight 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 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 second. 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 design and development, 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 so that if the system fails we will miss it or is it a result of a design decision that may be changed if necessary.

Requirement engineering/architect's vocabulary is very different from what is used in performance testing or capacity planning. Performance and scalability are often referred as examples of quality attributes (QA), a part of nonfunctional 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.

After 27 almost instantaneous responses, *there was a two-minute wait loop for the* 28th time for the same operation.

QA scenario defines source, stimulus, environment, artifact, response, and response measure[9]. For example, the scenario may be that users initiate 1,000 transactions per minute stochastically 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 stochastic 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 (Planning language) was suggested by Tom Gilb and may work better for quantifying quality requirements[10]. 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.

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 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, 80 percent below four seconds, 99.5 percent below six seconds, and 99.9 percent below 15 seconds (especially if we know that the difference in performance is defined by distribution of underlying data). Other examples may be 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 index) is a single metric of user satisfaction with the performance of enterprise applications. The Apdex metric is a number between zero and one, where zero means that no users were satisfied, and one 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 to mean 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 is important to trace requirements back to their source.

Requirements verification is checking if the system performs according to the requirements. To make meaningful comparison, both the requirements and results should use the same metrics. One consideration here is that load testing tools and many monitoring tools measure only server and network time. While end user response times, which business is interested in and 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 which is 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. But 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 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 track it down and fix 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.

Michael Bolton described this situation concisely[11]:

As Richard Feynman said in his appendix to the Rogers Commission Report on the Challenger space shuttle accident, when something is not what the design expected, it's a warning that something is wrong. "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." When a system is in an unpredicted state, it's also in an unpredictable state.

To summarize, 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.

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 something quantitative and measurable in the end. Making requirements too complicated may hurt. We need to find meaningful goals / requirements, not invent something just to satisfy a bureaucratic process.

If we define a performance goal as a point of reference, we can use it throughout the whole development cycle and testing process and track our progress from the performance engineering point of view. Tracking this metric in production will give us valuable feedback that can be used for future system releases.

REFERENCES

[1] Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE, 2004. http://www.computer.org/portal/web/swebok

[2] Barber, S. Get performance requirements right - think like a user, Compuware white paper, 2007. http://www.perftestplus.com/resources/requirements_with_compuware.pdf

[3] Sevcik, P. Using Apdex to Manage Performance, CMG, 2008. http://www.apdex.org/documents/Session318.0Sevcik.pdf

[4] Miller, R. B. Response time in user-system conversational transactions, In Proceedings of the AFIPS Fall Joint Computer Conference, 33, 1968, 267-277.

[5] Nielsen J. Response Times: The Three Important Limits, Excerpt from Chapter 5 of Usability Engineering, 1994. http://www.useit.com/papers/responsetime.html

[6] 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

[7] Sevcik, P. How Fast Is Fast Enough, Business Communications Review, March 2003, 8–9. http://www.bcr.com/architecture/network_forecasts%10sevcik/how_fast_is_fast_enough?_20030315225.htm

[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] Bass L., Clements P., Kazman R. Software Architecture in Practice, Addison-Wesley, 2003. http://etutorials.org/Programming/Software+architecture+in+practice,+second+edition

[10] Simmons E. Quantifying Quality Requirements Using Planguage, Quality Week, 2001. http://www.clearspecs.com/downloads/ClearSpecs20V01_Quantifying%2 Quality%20 Requirements.pdf

[11] Bolton M. More Stress, Less Distress, Better Software, November 2006. http://www.stickyminds.com/sitewide.asp?ObjectId=11536&Function=edetail&ObjectType=ART

About The Author

Alex Podelko *has specialized in performance engineering for the last fourteen years. Currently he is Consulting Member of Technical Staff at Oracle, responsible for performance testing and tuning of Hyperion products. Alex has more than 20 years of overall IT experience and holds a PhD in Computer Science from Gubkin University and an MBA from Bellevue University. Alex serves as a board director for Computer Measurement Group (CMG), His collection of performance-related links and documents can be found at http://www.alexanderpodelko.com/*