Use Case-Driven Performance Engineering without "Concurrent Users"

Morten Heine Sørensen
Formalit
Byenden 32, 4660 Store Heddinge
Denmark
+45 3031 2923
mhs@formalit.dk

ABSTRACT

The concept of *concurrent users* often causes confusion when used to define performance requirements in industrial software projects. The term is frequently used to state performance requirements without clarification of *what* the users will be doing, or *how often*. This paper offers a thorough analysis of the concept and related notions.

Despite the confusion surrounding it, the concept of concurrent users – in a precise form – is advocated in the community for stating performance requirements. However, we argue in this paper that, even when stated in precise terms, this approach has drawbacks. Indeed, a system may perform *better* than expected, even if the number of concurrent users it can handle is *worse* than expected. A better suited notion is that of *through-put.*

But even when basing performance requirements on clear, well-suited concepts, there appears to be no uniform format in the literature for such requirements. In particular, the requirements are sometimes stated in general, rather than for the specific areas of functionality of the system. As a consequence, the point may be missed that the through-put may be unevenly distributed over the functionality of the system. In this paper we therefore advocate the format of *performance-annotated use cases*, adding requirements on through-put and response-time to the traditional use case.

It is well-known how *functional* test cases are developed from use cases. In contrast, less has been said about the generation of performance test cases. Therefore, we show how the enriched use cases not only provide precise and meaningful requirements, but also yield detailed specification of the performance test set-up which can be directly input as configuration of load test clients. As a bonus, initial configuration of the system's capacity for handling concurrent users and requests is also provided.

Finally we outline an overall approach to performance test based on the above ideas. The approach has been followed in several industrial projects.

ICPE'13, April 21–24, 2013, Prague, Czech Republic.

Categories and Subject Descriptors

D.2.8 **[Software Engineering]** Metrics – Performance measures.

Keywords

Use cases, agile development, performance requirements, concurrent users, through-put, response time, think time, load test.

1. CONCURRENT USERS: CONFUSION

In a recent project in which the author participated, the four most central performance requirements were stated roughly as follows:

R1. The system must support 1500 concurrent user sessions.

R2. The response time for any user action must be at most 1 sec.

R3. The response time for a service call must be at most 0.1 sec.

R4. The system must handle 1000 events per day.

The requirements pertain to a system for which users can log in, perform user actions, and log out again. The system also receives service calls from external systems. Finally, events arrive at the system through either user actions or service calls, see Figure 1.

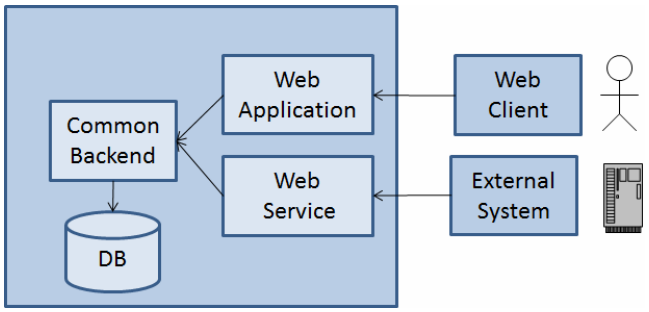

Figure 1. An interactive system also called by external systems.

Requirement **R1**, about *concurrent users,* is unsatisfactory because it does not state what the users do or how often they do it.

Handling 1500 users who log in and do nothing more is very different from accomodating the same number of users repeatedly requesting a heavy operation every second. A system may easily be capable of the former, while unable to live up to the latter. Whether a system satisfies the requirement is therefore heavily dependent upon interpretation.

Requirement **R2**, about *response time*, may be misleading because it fails to state that the response times must hold while the system is running at the expected *load*.

Responding promptly to a single user is an entirely different task than maintaining reasonable response-times for a large number of users simultaneously burdening the system.

Requirement **R3**, also about response time, is vague for the same reason as the second requirement: it fails to refer to the expected load.

In fact, both service call response times and response times experienced by users should respect the stated upper bounds while the system is burdened with the combined expected load from users *and* external systems, because both types of requests may share the same resources.

Finally, requirement **R4**, about *through-put,* is vague, because it does not state what functions are executed in the system at what frequency.

The event handling may have different flows, involving a different number of steps, depending on the type of event. Handling 1000 single-step flows may be a lot easier than 1000 flows each with 10 steps, and if the latter is closer to what actually happens when the system is in production, measuring the requirement in the former manner says little about what will happen during production.

Thus, the point is: *Requirements for response times are relative to some load; and if the load is defined in terms of concurrent users, it should explain what the users do and how often they do it.*

The above review of the requirements **R1**-**R4** concerns one specific project, of course, but the point is valid more generally. It frequently occurs in industrial projects that the client has difficulty expressing performance requirements in a precise manner, if they are stated at all.

This may be due to the fact that the performance requirements are on the border between the business domain and the technical domain. For instance, as will be argued in Section 2, the concept of concurrent users may be a purely technical term, or part business, part technical, depending on its precise understanding.

People writing requirements are often business people with limited training in IT, at least when there is a clear separation between the vendor and the client, and especially if the project is fully described by the client prior to one or more vendors making bids for the contract. These people often lack the necessary training or experience for stating precise performance requirements.

Another reason may be that there is little tradition for writing precise performance requirements in many organizations. As a consequence, the habit is not picked up by the participants of the organization's different projects.

As a further illustration, we refer to the standard contract ("K02") for IT projects delivering systems to public organizations in Denmark [18], see Figure 2.

K-1 The response times for any part or parts of the Delivery must comply with the following requirements:

Transaction	Description/if applicable assumptions	Service level goals for compliance rate, in %	Service level goals for maximum response time, in seconds
Simple	[...]	[...]	[...]
General	[...]	[...]	[...]
Complex	[...]	[...]	[...]
[If applicable, more specific transaction]	[...]	[...]	[...]
[...]	[...]	[...]	[...]

Figure 2. Performance requirements in K02.

Note that the requirement only states the maximum response time (last column) and the percentage of requests/transactions that must be below the maximum response time (third column).

The guide to the standard contract mentions the following example:

"Within any half-hour period, the response-time must be kept within the following limits:

- 99.5% of the response times must be less than 20 seconds.
- 98% of the response times must be less than 8 seconds.
- 95% of the response times must be less than 5 seconds.
- The average response times must be less than 3 seconds."

While very explicit on response times and fractiles, nothing is said about load. It must probably be understood that such details fall under "assumptions" in the second column.

A new version ("K03") has recently been developed, accommodating agile thinking [19]. However, the handling of performance requirements is not significantly changed.

It could be argued that load is only relevant for a particular kind of application – one used by multiple users simultaneously. For an application running entirely on a stand-alone PC making no request to other systems, the concept is not relevant and should therefore not be adopted in a standard for contracts covering all kinds of applications. However, the multi-user scenario is a very common (and the only one considered in this paper), and for this reason the argument is not convincing.

In order to revise the requirements **R1**-**R4** according to the point made earlier, we must define precisely the concepts of concurrent users, response time, and load – the latter will be rephrased as through-put. We begin with the service calls, because they constitute the simplest case.

Consider a software system responding to individual, unrelated requests; we denote this kind of system *session-less*. For instance, it could be an application server exposing web services.

Define the following notions:

- *Average response time* (ART): average time in minutes from the system receives a request, until the response is sent. This is a measure of the system's *speed*.[1]
- *Through-put* (TP): The number of requests finished in a minute. This is a measure of the system's *capacity*.
- *Concurrent requests* (CR): The number of requests being served by the system at any given moment.

The three notions are related to each other via the following rule, known as *Little's Law:*

$$TP = CR/ART.$$

Note in particular that if we know two of the values, we can compute the third.

Example 1. A system providing a public web service with currency exchange rates is observed over a minute to have a through-put of 60 requests and an average response time of 6 seconds (0.1 minute). It follows that, on average, there were 6 concurrent requests along the way, see Figure 3.

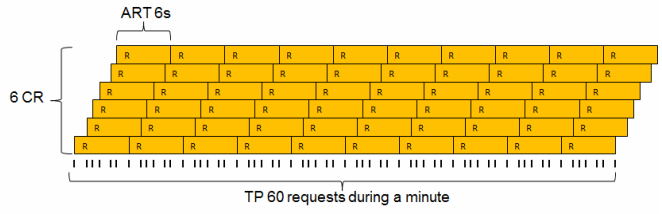

Figure 3. Little's law.

The small vertical bars at the bottom "|" indicate the arrival of requests. The solid yellow blocks indicate the time during which the system serves a request. Each level in the figure indicates a request waiting to complete or, equivalently, a server thread handling a request. At each level, whenever one request finishes, a new one is served.

Note that CR, like ART, is an *average* number. If, during the minute, the response times go up and down, so will the number of concurrent requests. In fact, it is unlikely that the number of concurrent requests will be precisely 6 all the time.

As mentioned, Little's law lets us compute one value from the two others. Another relationship between the three notions, may illustrate the concepts further. It concerns the situation where we test a system with a specified number of test clients that repeatedly fire requests—of course, this number is what we have called CU. If we begin with a small number, the system will probably have no difficulty serving the requests with some ART and TP. As we increase CU, initially ART will not be affected, but TP will grow, since we are getting more work done. At some point, the server's capacity for handling requests in parallel will be exhausted, and from this point on TP will, ideally remain the same, whereas ART will grow linearly, because the additional requests must now wait in line for their turn. The situation is illustrated in Figure 4.

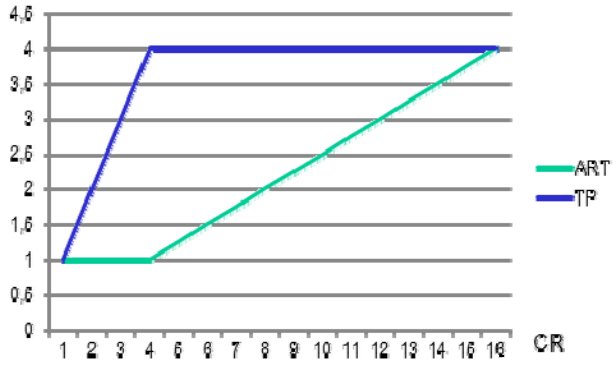

Figure 4. Effect on ART and TP of increasing CU.

In reality the graph may look different, because the queue mechanism often does not work in the ideal manner, see Figure 5.

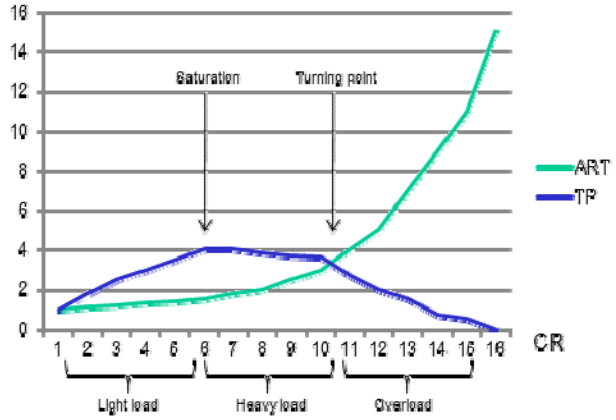

Figure 5. More realistic scenario.

For a heavily loaded server, it is an interesting exercise to configure the server's number of concurrent threads high enough to exploit the server's resources and low enough to avoid overloading the server (the additional requests will then wait in line in the server's execute queue, instead of burdening the server's resources).

Next consider a system against which a user logs on, performs a number of actions, and logs off again; we call this type of system *session-full*. The notion of logging in may not actually consist in providing identity and credentials; it could amount to an anonymous user having a session identified by a cookie returned by the system at the first request, and resubmitted in each subsequent request by the browser. Similarly, the notion of logging out may simply amount to the session timing out.

The concepts ART and TP carry over to the new setting, but CR is split into two separate concepts CU and CAU (the latter being the closest relative to CR); in addition a new notion concerning the user *think time* becomes relevant:

[1] A similar definition is to measure the time from a client makes a request until the client receives the response. This includes network time. During a performance test it may be relevant to ensure that the experienced network time is actually realistic. A further variation is to include also the client's time to render the response.

- *Concurrent logged in users* (CU): The number of users that have logged in, but not yet out. In other words, the number of users that have a *session* with the application.
- *Concurrent active users* (CAU): The number of users waiting for a response from the system at any given moment. In other words, the number of *threads* currently in use by the system to serve user requests.
- *Average think time* (ATT): The time taken from a user receives a response from the system until he or she sends the next request within the session. There is no think time after the last response is received; but if there is no explicit log out, the think time after the last step is considered to be the same as the time until the session times out.

Note that think time is meaningless in the setting of a session-less system. There is no concept of user, only individual requests, and each request is unrelated to all others. It is irrelevant whether subsequent requests are made from the same user/system.

What makes sense, though, is through-put – a certain number of requests are expected per minute. Moreover, we can discuss the *distribution* of the requests within the through-put. For instance, the through-put may be 120 requests per hour, but it could be that 90 requests are received in the first half hour and the rest in the second half hour.

When discussing a fixed through-put we always assume the distribution is more or less even (perhaps with some amount of randomization) within the period. Thus, if testing a system at peak load 120 requests per hour, where in reality the distribution is uneven as described above, we should rather test it at a peak load of 90 requests per half-hour, i.e. 180 requests per hour.

Coming back to the setting of session-full systems, the two new notions CU and ATT are related to TP and ART by the following generalization of Little's law, also called the *Response Time Law*:

$$TP = CU / (ART + ATT).$$

Example 2. A world-wide insurance company has a system for reporting damages online. A through-put of 8 reports per minute is observed at peak.

The customer logs in and reports the damage, and is automatically logged out at the end after the last step. The process has 5 user steps (log in, click "Report damage", enter details part I, enter details part II, confirm). The system has an average response time of 5 seconds for each request. Suppose that any user "thinks" 5 seconds before each next action; this includes the time to actually enter the information to the system.

Since the total ART for the whole report is 25 seconds and the total ATT is 20 seconds, we have a total ART + ATT of 45 seconds, i.e. 0.75 minutes, so CU is 6. The situation is illustrated in Figure 6.

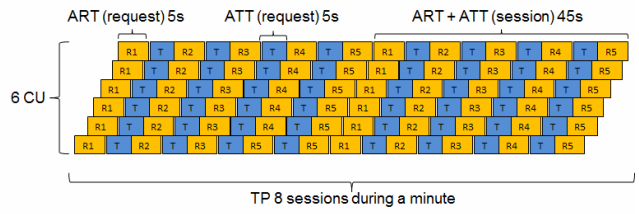

Figure 6. Response Time Law.

Each yellow block again represents a request, and now the number (R1-R5) indicates which request inside the flow is meant. The blue blocks indicate the user think time.

Note that TP on the one hand and ART plus ATT on the other hand must be understood relative to the same unit of work, which can be individual steps or the whole session. In the latter case ART and ATT is the sum of all ARTs and ATTs for the session.

Example 3. Counting requests instead of reports/sessions in the preceding example, we have 5 requests per report, i.e. a TP of 40 requests per minute. ART is 5 seconds. Since the user is immediately logged out after the last step, there is no think time after that step, but 5 seconds after the first four steps. This yields an ATT of 4 seconds per step. Thus ART + ATT is 9 seconds, i.e. 0.15 minutes. In conclusion, CU is again 6.

The density of active users within logged in users is given by:

$$CAU = CU \cdot ART / (ART+ATT).$$

In other words, the factor between CU and CAU is the time spent waiting for the system's response over the total time, i.e. user waiting time plus user thinking time (the time the user is idle before making the next request).

Example 4. Continuing the preceding example, recall that the total ART for the whole report was 25 seconds and the total ATT was 20 seconds. Finally CU was 6, so CAU is 3.3.

Note that ATT=0 if, and only if, CAU=CU. This happens if the session has just one step after which it immediately ends. This is what happens in the session-less case; so the latter can be considered a special case of the session-full case.

At any point, the active users are a subset of the logged in users which, in turn, are a subset of the *registered users,* if the latter concept makes sense for the system under consideration. It is not considered important whether users in subsequent sessions are actually the same or different registered users.

A few remarks about sessions may be in order. Above we assumed that users log in, do their work, and log out again. This type of session might be termed *dense,* since the user's activities happen in short time within the session, each step is separated in time from its predecessor by some small amount of think time.

However, not all systems with sessions work in this way. Another typical scenario is that users log in during the morning, do their work on and off during the day, and log out again in the evening. This type of session might be called *sparse*. In this scenario the number of concurrent users will typically be a fixed, known fraction of the registered users. There are even *ultra-sparse* sessions where the user can log-on, abandon the application, and hit it again weeks later with the session (basket) intact; for instance, Amazon has this nature. Probably, the session is passivated after a time-out and reactivated when the user returns.

There are two different ways to fit sparse sessions into the framework of dense sessions: we can either count sparse sessions as a variation of the dense sessions with very long thinking times, or we can separate the sparse sessions into sequences of dense sessions. We prefer the latter, though the choice should be made with some care.

The risk in splitting sessions is that the performance test may miss memory leaks, because the system terminates sessions quicker during the test than in reality, thereby yielding falsely low CU. Another risk is that response times may be higher than in reality,

because the caching behavior during the test may be unrealistic, because information cached in the session is lost during the test.

In fact, even in the case of session-less systems, it is important to reflect actual caching behavior, if present. Both cached information of relevance for all requests (e.g. reference data and other list of values) as well as information relevant only for individual users, but still stored in a general cache. (In the case of session-full systems, the latter kind of information can be stored in the session.)

We close this section with a reflection on the role of use cases in session-full systems. Recall that every user logs on, performs a number of steps, and logs out again. It is natural to envisage these steps as belonging to one or more use cases. The think time then amounts to the time the user waits within a use case, from one step to the next, after receiving a reply from the system, as well as the time from one use case ends, until the next is initiated. This is most natural when we work with dense sessions.

2. CONCURRENT USERS: CRITIQUE

We now have the apparatus to make the requirements of the preceding section more exact. But before doing so, let us turn to the literature for inspiration.

One precise form in the literature [17] is the following: "for use case ABC the system will respond to a valid user entry within 5 seconds for a median load of 250 [concurrent] active users and 2000 [concurrent] logged in users 95% of the time."

In the author's opinion, this formulation is still not optimal. First of all, a use case usually consists of several steps, and the response time must hold for each step in which an answer is expected from the system.

But, more seriously, it is in fact hard to justify an estimate on the number of concurrent *active* users. Of course, some kind of estimate is involved in any formulation of expected load, but the point is that the number of logged in and active users actually depend on internal attributes of the system itself.

Example 5. Consider two different systems implementing use case ABC mentioned above, and assume the use case has, say, 3 steps:

- System A has, in each step, a response time of 1 second as long as the number of concurrent active users is less than 200, but crashes when the number exceeds 200. It can handle 2000 concurrent logged in users.
- System B has, in each step, a response time of 5 seconds as long as the number of concurrent active users is less than 300, but crashes when the number of active users exceeds 300. It can handle 2000 concurrent logged in users.

Which system satisfies the requirement? Which system is the most desirable one?

Well, imagine the scenario that use case ABC is executed 1.000 times per minute by different users (it is not important if they are actually different from execution to execution of the use case).

If the users adopt system B, each execution involves 3 steps, each with 5 secs. spent by the system, and, say, 35 secs. spent by the user thinking before the next step. Thus each execution of the use case takes 2 minutes. With 1000 executions per minute, there will indeed be 2000 concurrent logged in users. The ratio between user waiting time and total time, i.e. user waiting time plus user think time is 5 to 5+35, i.e. 1 to 8, so among the 2000 logged in users, 250 will be active at any moment. Thus, our scenario corresponds exactly to the requirements and System B meets the criteria.

Now consider System A. It cannot handle the required 250 active users, but consider what happens in our scenario. Each execution of the scenario takes 3 times 1 + 35 secs, ie. 108 secs., i.e. 1.8 minutes. Thus there will only be 1800 logged in users, and the ratio between waiting time and total time is now 1/36, so there will only be around 50 active users at any point.

If we insist on verifying that System A can handle 250 concurrent active users, we have to push a through-put of 5.000 ABC executions per minute through the system, just because it handles requests more efficiently than System B (fewer if the average response time decreases with the increase in tested through-put).

The point is: *The faster the system, the fewer concurrent logged in and active users will be required to obtain a given through-put.*

Since we do not know the response time in advance, we do not know what to require of the system regarding concurrent logged in and active users.

In some situations, the number of logged in users may be fixed. For instance, all employees in a department may log in during the morning and log out in the afternoon. But for a fixed number of executions of use case ABC, the number of concurrent *active* users still depends on the response times of the system. Thus, again we cannot know the number of concurrent active users.

In some cases, the users of a site may abandon it, if it is slow, and this may provide some self-adjusting behavior, where CAU does not increase, despite higher ART than expected, because CU decreases. But of course, this is not a satisfactory way to satisfy the requirements (partially).

We conclude that it is difficult to predict CAU because it depends on ART, which we cannot know before the system is built. Will requests take half a second or a whole second? We cannot know.

In contrast, the through-put for each use case will often be known. For instance, a use case may concern the creation of some kind of business entity in the system, contracts, customers, orders, etc., and it might be known by the business how many of these arrive over a period. It is more difficult to predict the through-put of use cases that do not leave a trace that makes sense in business terms. For instance, if a user can browse entities before selecting one, it may be difficult to know the amount of browsing. In this case, some kind of estimate must be made.

Having fixed TP and ATT to some expected values, and fixed an upper bound for ART, we can compute an upper bound for CAU, and use this value for the relevant purposes. But one might wonder whether we have replaced one problem by a more difficult one: Is it not as difficult to predict ATT as ART? The answer to the last question may be "yes," but note that CAU is independent of ATT. Indeed, it follows from the response time law and the law regarding user density that

$$CAU = ART \cdot TP.$$

Intuitively, ATT does not impact CAU, because the user won't be counted as active during the think time anyway. (Note, incidentally, that the latter rule says that Little's law also holds for the session-full case, when we read CR as CAU.)

However, there are other reasons that it may be important to estimate ATT as precisely as possible. If we run a test without think time, despite having estimated ATT to some value, CU will be lower during the test than in reality, so we may miss memory problems stemming from many or large sessions. Also, varying think times in reality could cause chaotic phenomena causing difficulties for the system that are not revealed during the test. At least this is claimed in the literature.

A similar point pertains to the definition of which use cases a user executes in a scenario. Consider a system for paying bills online. It has use cases Log in, Pay Bill, Log out. In every execution, the user may repeat "Pay Bill" a number of times. The question now arises whether, for a given through-put, it makes any difference how it is distributed over users; that is, whether it makes any difference whether we have, say, 30 users each paying two bills (in a single session) or 60 users each paying one bill.

Example 6. Assume TP is indeed 60 bills paid per hour, i.e. 1 per minute. Users log on, pay one (or two) bills, and log out again. For simplicity we assume each of the three use cases has a single step only. Suppose ART is 5 seconds for of the three use cases and that ATT after Log on and Pay bill is 35 seconds.

In the one-bill per user variant we have ATT+ART = 85 seconds, i.e. 1.4 minutes, so CU = 1 · 1.4 = 1.4. CAU is 0.25.

In the two-bill per user variant, the through-put of sessions is 30 per minute, i.e. 0.5 per minute and ATT+ART=145 seconds, i.e. 2.4 minutes, and so CU = 0.5 · 2.4 = 1.2. CAU is 0.17.

The one-bill per user variant has slightly higher CU and CAU, because the overhead of use case Log On and Log Out is higher. In practice, such overhead may be negligible.

However, with each user paying two bills, there could be some benefit of caching, not experienced with separate users. If the application implements a relevant type of caching, this will be missed during the performance test when we run with one bill/per user, which may thus yield falsely high ART.

Above we have assumed that the system under consideration is some kind of multi-user application (where the users may be other systems). The notions of concurrent requests, logged in users, and active users usually will not be interesting for a desk-top application used by a single user.

Now consider a thick client, either the old-fashioned type (i.e. an application written in, say, C++, Visual Basic, or Java, that make server calls) or a modern Ajax-application running in a browser, with, say, HTTP calls mixed with significant portions of Javascript running entirely in the browser without service calls. In this case, only the functionality that actually makes server calls fits into the framework. In other words, the pure client functionality is not relevant for a load test of the server system.

3. ANNOTATED USE CASES

Recall that in Section 1 we complained that requirements R1 and R2 were vague because, although they specify restrictions on response time and the number of concurrent users, they do not explain *what* those users are expected to be doing or *how often*. We subsequently clarified the "how often" part, by introducing precise notions of concurrent user and related concepts. Also, we went on by arguing that through-put was a better suited notion for quantifying "how often."

It remains to address the "what" part. Indeed, even when basing performance requirements on clear, well-suited concepts, such as through-put, there appears to be no widely adopted format in the literature for relating these notions to functionality of the system. Perhaps this is the reason that requirements are sometimes stated in general for "the system", rather than for specific areas of functionality of the system. As a consequence, the point may be missed that the through-put may be unevenly distributed over the functionality of the system. Requirements R1 and R2 also illustrate this.

Use cases, *user stories*, and other similar techniques for specifying functional requirements are in wide-spread use today. They are ideally suited to support iterative development of a system, because the iteration plan can be arranged according to use cases. Each of the traditional disciplines – requirements, architecture, development, testing – can be carried out in each iteration for some subset of the use cases, and the iteration yields a complete version of the system, with functionality implemented end-to-end.

We therefore suggest the format of *performance-annotated use cases*, adding requirements on through-put and response-times to the traditional description of use cases. For instance, consider the use case description in Figure 7.

The use case allows a bank customer to pay bills from one of his accounts using an online banking system. The last three rows in the template describe *required through-put, required maximum response time,* and *assumed think time.* We discuss each in turn.

Use Case: Pay Bill		
Prerequisites	The customer is logged in.	
Standard flow	The customer	The system
	1. Selects Pay Bill.	2. Requests code line for bill, amount, account and date.
	3. Enters details and clicks OK.	4. Validates details, and requests password.
	5. Enters password and clicks Confirm.	6. Validates password, transfers amount, shows main menu again.
Alternate flows		
Result	Amount deducted from account and transferred to receiver.	
Through-put	3200 per hour (2 per user session).	
Response time	1 sec. for step 2 and 4; 10 secs. for step 6.	
Think time	10 secs before step 3 and 5.	

Figure 7. Performance-annotated use case.

The required through-put describes the through-put *for this particular use case* that the system must be able to support. In this example, the entry also mentions how many executions are contained in the same user session. The reason for this is that, due to caching behavior, it may make a difference to the system whether a single customer is paying several bills or each bill is paid by a separate customer in a separate session, as previously mentioned. The performance test should mimic the actual usage patterns of the system as closely as possible, taking time, money, risk and other relevant factors into account.

The required maximum response time specifies an upper bound for each system step that the system must observe. In a real

project, the system will probably only be required to meet the requirement in a specified percentage of the cases, say 95%. It may then be a requirement that in the remaining 5% of the cases, the response time must be at most, say, twice the specified maximum, i.e. 2 seconds after step 1 and 3, and 20 seconds after step 5. For instance, something like this is the case with the standard contract K02 as mentioned earlier.

Also, the actual maxima (1 and 10 seconds) may not be stated explicitly for each use case step; rather, the steps could be categorized into small, medium, and large, and a general maximum response time could then be formulated for each of the three types of step, say 1, 10, and 30 seconds, respectively.

The last entry describes the user think time. As previously explained, the think time may be significant because it drags sessions longer than what follows just from the response-times, thereby increasing the number of concurrent logged in users, and this could have an impact e.g. on memory consumption. In general it is desirable to reflect as closely as possible how the system is actually used in real life, though compromises may be made when balancing resources against risk. As with response times, the think times may be categorized into small, medium, and large (say 1, 10, and 60 seconds, respectively).

Say that we write every medium step with a single underline, a large step receives two underlines, and a small step receives no underline. Then a more quite notation for the above use case could be as shown in Figure 8.

Use Case: Pay Bill		
Prerequisites	The customer is logged in.	
Standard flow	The customer	The system
	1. Selects Pay Bill.	2. Requests code line for bill, amount, account and date.
	3. Enters details and clicks OK.	4. Validates details, and requests password.
	5. Enters password and clicks Confirm.	6. Validates password, transfers amount, shows main menu again.
Alternate flows		
Result	Amount deducted from account and transferred to receiver.	
Through-put	3200 per hour (2 per user session).	

Figure 8. More "quiet" notation.

The question arises what should be done about the alternate flows (or scenarios) of the use case. In practice, some prioritization must be made of use cases and their scenarios. It may easily be the case that not even all use cases are included in the performance test, let alone individual scenarios of a single use case. The decision must necessarily be decided on a case-by-case basis. For instance, if a use case concerns the possibility of doing searches in the system, and the alternate scenarios of the use case correspond to different ways of searching, then more (or all) of them could easily be relevant, so response times for all each of them should be specified. Fortunately, this is easy with the quiet notation.

Note that the use case refers to a *maximum* response time, rather than an *average* response time. Nevertheless, the response time law still provides valuable information. Indeed, if the response time for every system step is *at most* a specified amount, then the *average* response time will be less than this amount, and therefore the through-put will be *at least* the amount specified by the response time law, for a given number of logged in users.

Similarly to the performance requirements for use cases, we may also have performance requirements regarding maximum response time and minimum through-put for web services called externally. We do not have a specific format for this in mind.

4. PERFORMANCE TEST CASES

As indicated earlier, there is a widely adopted practice of generating functional test cases from use cases. Roughly, the test cases can be seen as instantiations of use case scenarios where concrete values are substituted for named concepts.

For instance, whereas a use cases speaks of *a customer* who logs on and pays *a bill* of *some amount*, the test case will involve a *specific customer* known to the system who will pay a *specific bill*, known to the system, of a specified, concrete amount.

These test cases could in principle be extended with information about expected response-time and assumed think time, as well as expected through-put. However, in reality such test cases will not be run repeatedly, manually by humans. Instead they are typically run *once* and recorded with some proxy and then revised so as to use input data from files or a database. After this the test cases can be run automatically by a load test client. Figure 9 below has (in the left pane) a test scenario that amounts to the main success scenario of a the Report Damage use case of the insurance system mentioned earlier.

The test cases are then configured with respect to the actual performance parameters. For instance, in the popular tool *JMeter*, one configures a so-called thread group for each test case, and part of the configuration is to fix a number of threads (users), see Figure 9.

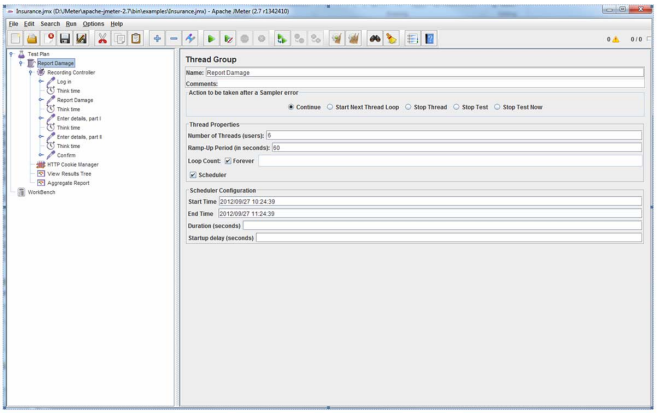

Figure 9. Concurrent logged in users.

This number corresponds to CU and can be computed from the required through-put and maximum response time, plus the assumed average think time. For instance, in the example with the insurance company, we need 6 threads, as mentioned in the last example, if the system precisely meets the requirement for ART.

If more threads turn out to be needed, it must be because the system cannot meet the requirement for ART. If the system's response time is significantly lower than the required ART, we may actually need fewer threads than 6.

In fact, one can configure JMeter to achieve the required TP by using the necessary number of threads (up to 6), whatever that number turns out to be depending on the system's response time behavior, see Figure 10.

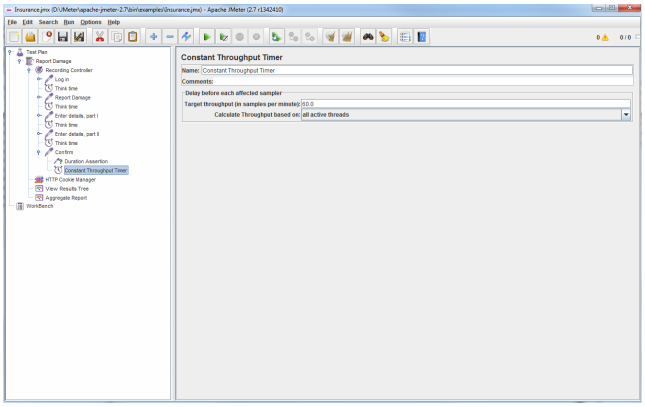

Figure 10. Achieving the required through-put.

In addition to CU we can configure think time, see Figure 11.

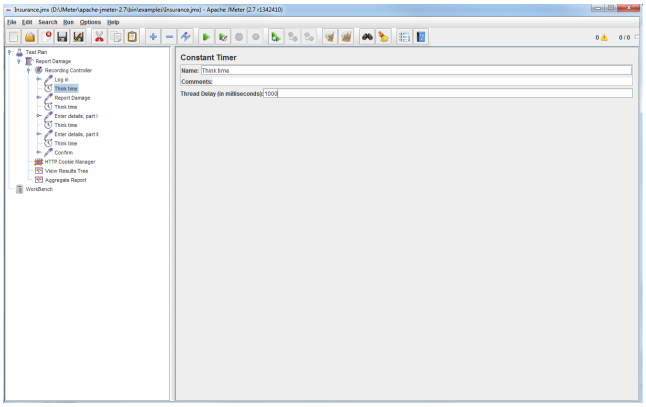

Figure 11. Think time.

Finally we can mark excessive response-times as failures, see Figure 12.

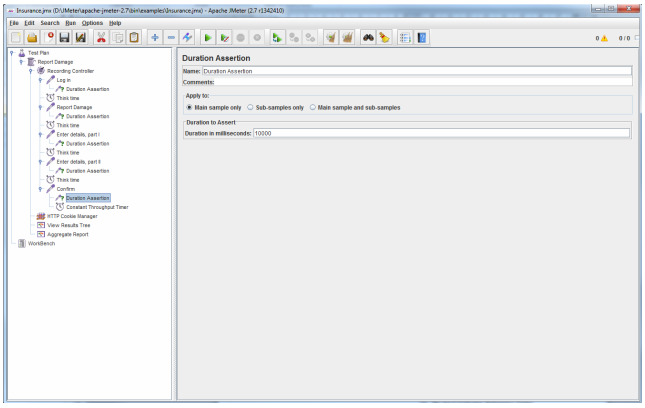

Figure 12. Marking high response time as failure.

Finally, we can ask the tool to produce a report for the test run that displays the response times and through-put, so that we can see whether the requirements are met. See Figure 13.

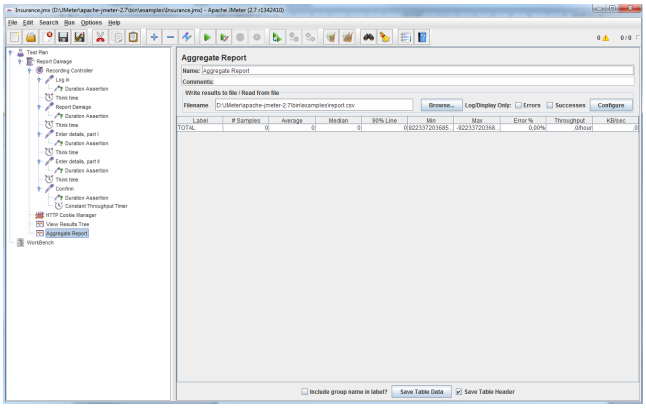

Figure 13. The test run results.

Thus we see that a typical load test tool, exemplified by JMeter, provides facilities for configuring test cases and performance requirements expressed in terms of through-put, response time and think time.

5. OVERALL APPROACH

We finally summarize our approach to performance-testing which comprises a number of activities; these happen partly before we go into the iterations of a project, and partly inside the iterations themselves.

1. Identify the functional requirements.
2. Identify the system and environment.
3. Identify the performance requirements.
4. Prioritize the performance requirements.
5. Choose performance test type.
6. Plan the performance test.
7. Build the performance test.
8. Run the performance test.
9. Analyze the output from performance test.
10. Profile the system.
11. Optimize the system.

In the first activity we identify the functional requirements, i.e. the use cases, or something similar. One may wonder why it is relevant to identify the *functional* requirements, when we are in fact interested in *performance,* i.e. *non-functional* requirements. The answer is that the latter pertain to the former; that is, each use case has its own through-put (and possibly response time) requirements.

In the second activity we get an overview of the architecture of the system and the production environment as well as the performance test environment. It is important to understand the architecture, because this has a bearing on how we formulate and test the performance requirements. For instance, we need to know

if the system can be accessed both interactively and through web service calls.

But the architecture is also important for deciding what we measure during the test. For instance, a database server calls for a different kind of attention as compared to an application server. Finally, the environments are important because it should be verified that the performance test environment resembles the production environment as much possible.

The configuration of *threads* through-out the production environment can be based (at least in the first version) on the performance requirements.

Example 7. The insurance company mentioned earlier is building a new system for reporting damages online. The system must support a through-put of 60 reports per hour, and these are submitted in the Report Damage use case. The total maximum permitted response time for the whole use case is 25 seconds and the total think time is 20 seconds, so we have a total, so CU will be at most 45, and CAU will be at most 25. It follows that, in the worst case, the system must support 45 user sessions and 25 simultaneous requests.

If each request on the web server may open at most two connections to the back-end server, then a total of 50 threads will be needed at that level. If each thread on the back-end server, in turn, requires at most two connections to the database server, a total of at most 100 will be needed at that level. Here we have only considered the traffic from the online users. To all this, the threads handling web service requests must be added.

Moving on, we identify, in the third activity, the performance requirements, if they exist, or else contribute to their formulation.

In the fourth activity we decide which performance requirements to include in the performance test, as all can probably not be tested. Or, more concretely, we decide which use cases (and which scenarios of these) must have their performance requirements tested.

In the fifth activity we identify which type of performance test to run, e.g. load test, stress test, soak test, etc. (We omit detailed discussion of the different possibilities, as it is not the main concern in this paper.)

In the sixth activity we plan the performance test. The overall idea is that the performance testing of use cases follows the ordinary testing of use cases, i.e. it occurs in the iterations where the coding for those use cases is accomplished.

In the seventh step we build the test, as indicated in the preceding section. Of course, much scripting is usually needed.

In the eighth activity we run the test and collect statistics along the way. The performance test tool provides details about through-put and response time, as also shown in the preceding section.

In the ninth activity we analyze the output of the test and conclude whether the requirements are actually met.

If not, the tenth activity may include further profiling of the system using various tools to understand e.g. where excessive CPU time or memory consumption occurs in the system.

Once the reasons for inefficiencies are located, the system is improved in the eleventh activity.

Of course, the last four steps are often iterated several times.

6. CONCLUSION

We have provided an analysis of the fundamental concepts relevant for performance testing of multi-user applications: concurrent users, response-time, through-put and think time.

Also, we have criticized the user of "concurrent users" for stating performance test requirements as this notion is a technical attribute of the resulting system, which cannot be known in advance. Instead we have advocated the user of through-put.

We have also demonstrated how performance test requirements can be stated in a compact fashion adding just a few details to the existing description of use cases.

Moreover, we have shown where the identified performance test parameters fit into the construction of an automated performance test using a popular tool.

Finally we have fitted all the pieces into an overall performance testing methodology.

The paper draws on a number of references.

The exposition of fundamental laws connecting the different concepts related to performance requirements is inspired by [9].

As mentioned earlier, the critique of "concurrent users" was inspired by an example in [17].

The positioning of the paper inside a broader tradition of performance *engineering*, rather than merely performance *testing*, is inspired by [14]. Many other text books on performance testing emphasize not only the actual testing activities, but also the precise formulation of performance requirements as well as an overall approach for managing performance, see, e.g. [4], [7], [11] [12] [16].

Nevertheless, the literature distinguishes between performance *testing* and performance *tuning*, see [8], [13] [15] for some popular references in the latter category. The distinction is the same as that between testing for functional errors on the one hand, and fixing code bugs on the other hand. For functional testing, the distinction is often quite sharp in projects between those who look for errors and those who fix them. In the case of performance testing, the same persons are often involved in both activities

Use cases were invented by Ivar Jacobsen and popularized by Cockburn [1] and many others. They are in wide-spread use today.

A less formal approach is the user of *user stories,* see e.g. [2]. These contain fewer details and correspond more (but not exactly) to scenarios of a use case than to the use case itself. Performance requirements may also be attributed to user stories, though the individual steps of the scenarios of use cases may be useful for stating response and think times.

Developing test cases from use cases is a standard practice, see, for instance, [3][5]. It is natural to use the same approach to drive the testing of performance requirements and the idea occurs more or less explicitly in [10] and [17], and probably many other places. The guide for "K03" [19] also mentions the idea informally. Instrumenting use cases with response time requirements is hinted at in [10].

Our examples used the popular, open source tool JMeter [6]. There are numerous commercial tools available as well, as a quick Google search will reveal, but JMeter has also been used for many large-scale industrial projects.

7. REFERENCES

[1] Cockburn, A. 2001. *Writing Effective use Cases.* The Agile Software Development Series. Addison-Wesley.

[2] Cohn, M. 2004. *User Stories Applied for Agile Software Development.* The Addison-Wesley Signature Series. Addison-Wesley.

[3] Collard, R.1999. Test Design: Developing Test Cases from Use Cases. Software Testing & Quality Engineering Magazine.

[4] Haines, S. 2006. Java EE 5 Performance Management and Optimization. Apress.

[5] Heumann, J. 2002. *Generating Test Cases from Use Cases*. Journal of Software Testing Professionals.

[6] JMeter documentation. http://jmeter.apache.org/

[7] Joines, S., Willenborg. R., and Hygh, K. 2002. *Performance Analysis for Java Web Sites.* Addison Wesley.

[8] Killelea, P. 1998. *Web Performance Tuning.* O'Reilly.

[9] Lazowska, E.D., Zahorjan, J., Scott Graham, G. and Sevcik, K.C. 1984. *Quantitative System Performance. Computer System Analysis Using Queueing Network Models.* Prentice-Hall.

[10] Leffingwell, D. and Widrig, D. 2003. *Managing Software Requirements – A Use Case Approach*. Second Edition. Object Technology Series. Addison-Wesley.

[11] Microsoft Corporation. 2007. *Performance Testing Guidance for Web Applications.* Microsoft Press.

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

[13] Shirazi, J. 2003. *Java Performance Tuning.* O'Reilly.

[14] Smith, C. U. 1990. Performance Engineering of Software Systems. Addison-Wesley.

[15] Tow, D. 2003. *SQL Tuning.*O'Reilly.

[16] Zadrozny, P. 2003. *J2EE Performance Testing.* Apress

[17] Wikipedia entry on performance engineering, http://en.wikipedia.org/wiki/Performance_engineering.

[18] *K02 – Standard contract for long-term IT project.* http://www.digst.dk/Styring/Standardkontrakter/K02-Standardkontrakt-for-laengerevarende-it-projekter.

[19] *K03 - Standard contract for agile IT project.* http://www.digst.dk/Styring/Standardkontrakter/K03-Standardkontrakt-for-agile-projekter