Experiences in Teaching Quality Attribute Scenarios

Ewan Tempero
Department of Computer Science
University of Auckland
Auckland, New Zealand
ewan-at-cs.auckland.ac.nz

Abstract

The concept of the *quality attribute scenario* was introduced in 2003 to support the development of software architectures. This concept is useful because it provides an operational means to represent the quality requirements of a system. It also provides a more concrete basis with which to teach software architecture. Teaching this concept however has some unexpected issues. In this paper, I present my experiences of teaching quality attribute scenarios and outline Bus Tracker, a case study I have developed to support my teaching.

1 Introduction

It has long been understood that choice of architecture can affect the success of a system (Garlan et al. 1995, Shaw & Garlan 1996). There has been much work done to find ways to efficiently find the right architecture for a given set of requirements. One particular issue is how to establish that a proposed architecture does indeed match the requirements. Doing so requires that the relevant requirements be expressed in an *operational* form that allows an objective assessment of the fit of a given architecture. In the second edition of their book on software architecture, Bass et al. introduced the concept of the *quality attribute scenario* (QAS) (Bass et al. 2003). This concept can be used to express the (non-functional) quality requirements of a system in an operational form.

When I encountered the QAS concept, I thought it was a solution to a problem I faced. In the previous year, I had taught a section on software architecture, and have been unsatisfied with the answers I could give to the question "How do we know when we've got the right architecture?" When I discovered QAS, I immediately saw the potential for providing an objective means to answering the question. There was still the question of how to find the right architecture, but at least having established the QASs for a system we would know what the target looked like. What I didn't anticipate, however, was how difficult a concept it was to grasp for inexperienced software engineers. I speculate that it was easier for me as I had some understanding of what the issues were and so perhaps could more easily see how QAS would help. Prototypical software engineers however are mostly unaware of the concept of software architecture itself, never mind the issues surrounding the development of one. Nevertheless I still had to teach the concept to them!

In this paper, I discuss the problems that students experienced in learning about QAS and outline the **Bus Tracker** case study I have developed and use to support my teaching of software architecture, and QAS in particular.

The rest of this paper is organised as follows. In the next section I will describe the context in which I teach software architecture, so that others can determine how applicable my experience may be to their own situation. In section 3 I will briefly present the quality attribute scenario concept and discuss other related work. Section 4 presents the Bus Tracker case study. Section 5 presents the issues that I have observed students having when learning about QAS, giving examples from the class. Section 6 completes the Bus Tracker case study by providing a set of scenarios for it. I make some concluding comments in section 7.

2 Context

The material described in this paper is taught as part of a Software Engineering (SE) specialisation in the Bachelor of Engineering (BE) degree offered by the School of Engineering at the University of Auckland.

The BE degree is a four year undergraduate programme. Students who are accepted into the programme do a common first year consisting roughly of courses representing all the specialisations in the degree. At the end of the first year, students then apply to enter their specialisation of choice. Each specialisation then consists of 3 years of courses specific to that programme, although each year also has "professional development" courses covering such topics as communication skills, engineering management, ethics, and sustainability.

The SE programme has a course titled "Software Architecture" in the second semester of the second specialisation year (third year of the BE). At the point the students take this course, they would have done one course during the common first year that introduces fundamental programming concepts and a year and half (that is, 3 semesters) of (mostly) SE speciality courses. This includes, during their second year, courses covering topics such as object-oriented design, design patterns, data structures, computer organisation, quality assurance, discrete mathematics, statistics, probability, and a project course. During the first semester of their third year, SE students have courses on databases, human-computer interaction, and computer architecture. While they are taking software architecture they are also taking courses on networks and operating systems, as well as a project course. So the students have seen a number of related topics, however they have not really seen anything where the software architecture of anything is discussed in detail.

The "Software Architecture" course is logically divided into two parts. The first part is "middleware", covering such topics as remote procedure call and replication strategies and technologies such as RMI and CORBA. The second part, which I teach, is software architecture fundamentals. The organisation has come about due to curriculum realities and pedagogical decisions.

We, along with all other curriculum designers, have discovered that 4 years is not enough to teach everything

that we think is important. We have had to make compromises based on our teaching environment, available expertise, local demand, and the usual vocal individuals (Gruba et al. 2004), meaning we could not have two separate courses on middleware and software architecture.

Combining these two areas into one course is not unreasonable as there is a clear relationship between the two. In fact combining middleware with software architecture helped with a problem I had encountered when first teaching this course. Students in their third year of university generally have had little experience with large software systems and so have difficulty appreciating discussions on the architecture of systems. They have also had little experience with quality attributes such as reliability and availability, both of which are important to middleware discussions. By following on from the middleware part, I have some nice examples on which to based discussion on software architecture. The support provided by middleware on teaching software architecture has been noted by others (Royce et al. 1994).

While combining with middleware was useful, it still left me with only 6 weeks in which to cram (what I considered to be) the important fundamental concepts of software architecture. I had originally used the first edition of Bass, Clements, and Kazman's *Software Architecture in Practice*, but as I indicated in the introduction, was unhappy with my ability to answer what I considered to be an important question of any architecture, namely "is it the right one?" When the second edition came out (Bass et al. 2003) and introduced the *quality attribute scenario* concept, I saw it as the answer to my problem. I immediately adopted it and structured the rest of the material around it.

3 Background and Related Work

In this section I give a summary of the concepts needed for the rest of the presentation, and discuss related work. Most of the specifics here come from Bass et al. (Bass et al. 2003)

3.1 Software Architecture Concepts

As many have observed, there are a number of definitions for software architecture (SEI 2007). The specific definition I use in the course is not so relevant to this paper, but for completeness sake it is the one from Bass et al., namely:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

What is not obvious from this definition, although it is made clear in the text, is that what dictates a given architecture is not the functional requirements of a system, but what are often referred to as the non-functional requirements, that is, such things as performance, reliability, extensibility, and so on. These are what are now generally referred to as *quality attributes*. Since these impact choice of architecture, we need to be able to specify them in an operational way in order to be able to determine if the correct choice of architecture has been made. It is to do this that Bass et al. introduce *quality attribute scenarios* (QAS).

Two key points about quality attribute scenarios is that they are intended to capture *requirements*, and so it is not appropriate that they contain any kind of decision regarding how the system may be built, and that they are intended to be used to determine if a given architecture meets those requirements. The second point dictates what kind of information is needed in their description whereas the first dictates what kind of information should not be in their description.

Source	some entity that generates a stimulus
Stimulus	a condition or event that needs to be considered
Artifact	the thing that is stimulated
Environment	conditions under which the stimulus occurs
Response	what the artifact should do on arrival of the stimulus
Measure	how to measure the response to determine it is satisfactory

Figure 1: Quality Attribute Scenario parts (*From Bass et al.*)

Bass et al. identify six "system" quality attributes relevant to software architecture: performance, modifiability, availability, security, usability, and testability. The other commonly discussed quality attributes can be expressed in terms of these six (for example, portability is a specialised form of modifiability, and data integrity is a specialised form of security). There are other aspects that may affect choice of architecture, such as time to market or system lifetime, but it is these six that are my main focus for the use of QASs.

A quality attribute scenario consists of the parts shown in figure 1. It is the "measure" that makes a scenario operational. It is not enough to declare that a web server must be "fast" as we don't know what "fast" means. To some people this may mean generating a response in 100 milliseconds, but others may be content with a response in under 2 seconds.

The "measure" part dictates what is acceptable. But just having the response time is not enough information to evaluate an architecture. For example, does the 2 seconds apply to how long the server has to generate a response from when the request is received, or does it apply to the time between when the user clicks the submit button and sees a page displayed — these two situations involve different "source" and "response" values.

The time the user has to wait for a response depends on factors such as the network performance — the system can hardly be held accountable if the network fails. The context in which the response measure applies is described in the "environment" part.

There can be potentially many scenarios for a given system and so a concern is that some may be missed. Bass et al. distinguish between *general* and *concrete* QASs. General QASs are system independent; the same scenario can apply to many different systems. Bass et al. provide initial lists of possible values for each of the parts of a QAS for each of the system quality attributes. The process of finding QASs begins by choosing relevant values from these lists to identify the appropriate general scenarios. Then each general scenario is instantiated for the particular system under consideration by deciding on system-specific values for each of the general values to get concrete scenarios. One general QAS may result in many concrete QASs. Figure 2 shows the possible values for Performance general scenarios as described by Bass et al.

While I present the general scenario concept, and insist that students at least indicate which values they are using when developing concrete scenarios, I do not regard this concept as a complete solution. So, for example, I make it clear that we should not regard that all the general values provided Bass et al. are a complete set, or even that we must pick exactly one value from each list. General scenarios are (merely) a tool that help us produce concrete scenarios.

3.2 Related Work

Shaw and Clements have argued that software architecture is in its "golden age" and in the near future will reach

source of stimulus	an independent source (possibly within the system)
stimulus	periodic events arrive; sporadic events arrive; stochastic events arrive
environment	normal mode; overload mode;
artifact	system
response	processes stimuli; changes level of service
response measure	latency, deadline, throughput, jitter, miss rate, data loss

Figure 2: Possible general values for Performance (*From Bass et al.*)

the point of being an "unexceptional, essential part of software system building — taken for granted, employed without fanfare, and assumed as a natural base for further progress" (Shaw & Clements 2006). I would argue that to attain this status it needs to be part of any software engineering curricula (and possibly computer science curricula as well). As I have already suggested, and as others have confirmed, there are issues in teaching software architecture, especially to inexperienced undergraduate students.

To my knowledge, there has been no discussion of the teaching of quality attribute scenarios, but there have been some discussions relating to teaching software architecture in general. One of the first was by Royce et al. who discussed the use of a middleware product to teach software architecture at a graduate level (Royce et al. 1994). Their proposed course emphasised developing large-scale systems from reliable, pre-integrated, reusable components, and considered the ability to compare architectures. At the time of publication the course had only been offered once but the authors observed that it was complicated teaching software architecture concepts to students with little practical experience.

In contrast, Bucci et al. explained how they introduced the concept of software architecture early in the curriculum (Bucci et al. 1998). They focused on the view of software presented to developers (or students) by the tools that are used, and argued that tools that provided an architecture-level view of software would help students understand software architecture. While the part of the course I teach involves no actual writing of code, Bucci et al.'s point that students need the right *mental models* is, I believe, at the heart of the teaching software architecture.

More recently Lago and van Vliet discussed their experiences teaching two software architecture courses at the Masters level (Lago & van Vliet 2005). Their course goals included generating alternative architectures, describing architectures, and evaluating architectures, and they were particularly interested in the need to trade-off different stakeholder requirements and consequently the need to communicate effectively with stakeholders. I see the use of QASs as a key ingredient in such communication, as well as evaluating the final result. Lago and van Vliet do not mention QAS explicitly, although they use Bass et al. as the text for one of the courses. They do note the importance of the quality of "scenarios," but it is not clear if they mean QASs specifically, or a more general use of the term.

Karam et al describe their undergraduate presentation of software architecture taught at about the same level as my course (Karam et al. 2004). They present many of the same topics that I do, although they do no explicitly mention QAS. They claim that having complete executable examples allow the students to understand the material better. I am not convinced. As I will discuss later, I have found students tend to want to dive into implementation as soon as possible, even when they are trying to determine requirements. I am concerned that working with executable systems would reinforce that behaviour so my preference is to not deal with actual code.

4 Bus Tracker

The example I use in the case study was initially set as an assignment (see section 4.1). It involves development of a system for providing electronic display of estimated arrival times of public buses for a city council

4.1 History

This example was used as an assignment in the first offering of the course in 2002. The assignment was to "develop an architecture for the system." It was not used in 2003 (when QASs were introduced) but then reused in 2004 as the basis for the two-assignment sequence (see section 4.4). It worked so well for the assignment that in 2005 I decided to begin developing it as a case study for use in class and tutorials. The case study was then developed over the next 3 deliveries of the course.

4.2 Description

Figure 3 shows some of the initial details that are given to the students (the explanation at the bottom has been added for this presentation). Some other details are also provided, partly to provide some more concreteness to the exercise. For example, some of the functionality is described in more detail, such as what should appear on a display. Numbers are also given for how many bus stops and buses may need to be considered, and some possible performance characteristics of the communications systems and other hardware.

Nevertheless there are still many details that might be important to know when developing an architecture. For example, nothing is said about what hardware is available (some Computer Systems Engineers take the course each year and they have observed that different chipsets could be used for the bus subsystems with different capabilities and different costs), or how the estimates are produced. Partly this is due to my lack of knowledge in such things, and anyway such details also should not be so relevant to developing the quality requirements. But I use this lack of information to make the point that as software architects the students are likely to be in the same situation (not having complete information) and they are welcome to make up any details they feel are necessary so long as they can justify their choices and the choices do not make the exercise trivial.

In fact, the details I do give them are not totally consistent, or not that relevant, or not actually likely to be what the client actually wants, as I will discuss later. Over the years I have resisted the temptation to "improve" the information. Information provided by clients is notorious for not necessarily being of high quality and others have observed that development of the architecture is sometimes where the true requirements are determined (Bass et al. 2003, sidebar, p27). This is a point I can make more easily using the existing description than if I had provided a "sanitised" description.

Another aspect of this example that has proven useful is that it naturally decomposes into three subsystems: the bus subsystem, the display subsystem, and the rest (which I typically refer to as the "central" subsystem, although there is nothing implied in the description that that part has to be all in one place). This provides a nice example of how the system architecture can partially impact the choice of software architecture, as well as the distinction between system and software architecture.

As well as the aspects I've mentioned above, the Bus Tracker system is useful for the number of architectural questions that arise. In particular, there are are multiple examples of all the quality attributes, all leading to many interesting different possible scenarios. I will discuss some of these in section 6.

ARC wants a system called Bus Tracker that tracks buses. It wants to add GPS to all of its buses so that it can track where they are to within 100 metres. They will use this information to provide estimated arrival times of buses at each major bus stop.
The unit to be placed on each bus consists of a Global Positioning System (GPS) receiver, a radio transmitter, and other bits of hardware and software. The GPS receiver can can determine its position to within 10 metres at each second. If calibrated properly, it can reliably track the bus position and speed throughout the journey. It transmits this information, along with the bus' identifier to the Bus Tracker system on a regular basis.
The major bus stops are where a number of bus routes converge. There are typically 20 or more buses that stop at these stops during peak travel times. The planned displays will have a radio receiver and room for display four or five bus numbers and times (that is, similar to those that already exist for the LINK bus).
The displays should repeatedly scroll through all the buses whose estimated or scheduled arrival times at that stop are sometime in the next hour. Once the bus is within 1 kilometre (that is, about 2 bus stops away) of a display, the estimated arrival time should be within 2 minutes of the actual time, 95% the time. All other displays should show a "best effort" estimated time.
The bus company also would like to allow bus users to get estimated arrival times for all buses at all times via their web site, and also via phone.

(The ARC — Auckland Regional Council — is an elected local government authority covering the Auckland Region and the cities within it with regulatory power and funding capabilities for such things as public transport, environmental protection and regional parks. The LINK bus is a particular bus service.)

Figure 3: Bus Tracker initial description.

4.3 Presentation

The way the case study is presented has evolved over the last 3 or so years. What is described here is the presentation used in the 2007 second semester (July-October) offering.

The first step is a tutorial session in which the students work in teams of about 5 members (self-formed) to identify the "non-functional" requirements likely to be relevant to the text as given. The purpose of this tutorial is to demonstrate the issues with specifying such requirements in a manner that allows for proposed architectures to be evaluated. It also has the benefit of giving the students a chance to come to grips with the Bus Tracker system itself, without the distraction of having to apply new concepts at the same time. Teams (the number depending on the amount of time available) are then asked to present one non-functional requirement to the class, and we discuss how useful their description is with respect to being able to determine if a proposed architecture meets the requirement.

At the time of this tutorial, the students have not seen the QAS concept, but they have had explained to them the general idea of what software architecture means and why architecture is important, and have seen fairly high-level descriptions of some software architecture examples.

The second step is a tutorial a week later. By this time the students have seen QAS, including general scenarios. In this tutorial, they are formed into teams (this time not of their choosing) and asked to develop two performance scenarios for Bus Trucker. Once all the teams have at least one completed (typically after about 30 minutes), one team is picked to present one of their scenarios to the rest of the class. This scenario is then critiqued.

Later on the same day, the course has a scheduled lab. We use this lab session to refine the scenario descriptions, with each team entering their descriptions into the SE Wiki.

In the next lecture session we go through several submitted scenarios for another round of feedback. Finally, in a following lecture, I present some scenarios of my own and explain my reasoning for choosing (or rejecting) them.

Although not directly relevant to this paper, the Bus Tracker system is later used as the basis for tutorials, and lab exercises on the development of structures for architectures.

4.4 Assignments

It is worth noting that the assignments follow the same pattern. Initially, students given a piece of text at a similar level of detail as in figure 3. Their first assignment is then to develop some number (typically 3) of quality attribute scenarios for a quality attribute (typically performance or availability) for the proposed system, and also one structure description that relates to one of their scenarios. These assignments are marked via peer assessment using an on-line peer assessment system (Hamer et al. 2007).

For the second assignment, I give them 2-3 scenarios and ask them to develop an architecture that meets those requirements. Doing this means that they get to see another set of scenarios I have developed for another system that they have had to think about, reinforcing the sequence from Bus Tracker. They submit their architecture description and justify it with respect to the scenarios. As this assignment is due only at the end of the teaching period, it is assessed in the more traditional fashion.

5 Issues

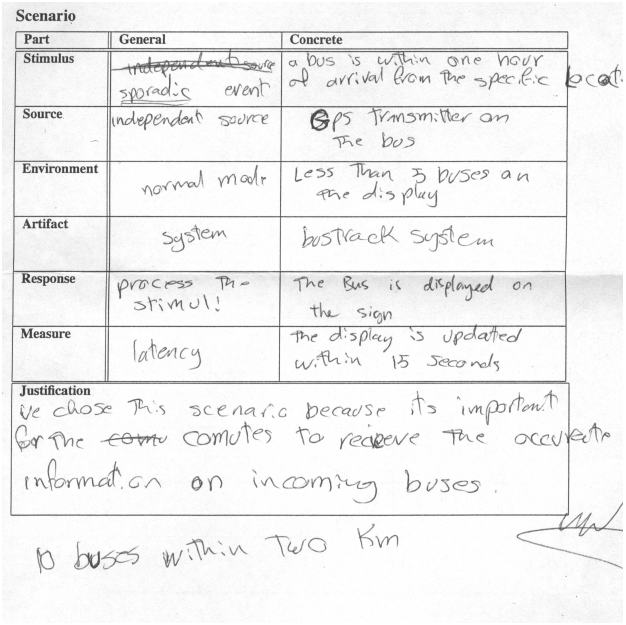
In this section I discuss the kinds of issues I have observed.

5.1 Overview

There are four areas of confusion that need to be addressed — what constitutes reasonable values for each part, how the values for each part interact, whether the choice of values constitutes a quality requirement, and whether the resulting scenario is relevant to the system being developed. Teaching QASs requires progressing through each of these areas. The first two areas are about how to construct a valid scenario, whereas the last two are about whether the scenarios are describing something useful.

Determining reasonable values is about such things as describing something for the stimulus part that really is a stimulus, or an artifact that really is a part of the system that is relevant to the creation of the architecture. The first scenarios produced by the students usually contain values that are somewhat acceptable, but lack the precision needed for the ultimate use of QASs — evaluating an architecture. However in some cases the values are not acceptable. A common example is the statement for the measure that contains nothing measurable

While it might be expected that the general scenario values would help reduce these problems, my experience is that early on, students have few problems creating consistent general scenarios but have difficulty with concrete scenarios. I speculate that this is because the production of a general scenario can be done by just choosing values from lists. This means a consistent general scenario can be produced without a great deal of understanding of what the individual values mean. A common mistake is to produce an acceptable general scenario, but then choose values for the concrete scenario that are inconsistent with the



Scenario

Part	General	Concrete
Stimulus	~~independent source~~ sporadic event	a bus is within one hour of arrival from the specific locat
Source	independant source	Gps transmitter on the bus
Environment	normal mode	Less than 5 buses on the display
Artifact	system	bustrack system
Response	process the stimul!	The Bus is displayed on the sign
Measure	latency	the display is updated within 15 seconds

Justification

We chose this scenario because its important for the ~~comp~~ computes to receive the accurate information on incoming buses.

10 buses within Two Km

Figure 4: A typical first QAS. (Most details are reproduced in the main text.)

chosen general values. For example, choosing "latency" as the general performance measure, but then specifying a concrete measure that is not latency.

The next area is choosing values that work together to describe a scenario. Here the issue is whether, for example, the stated stimulus can be produced by the stated source, and will be felt by the stated artifact, or whether the stated measure does measure something related to the stated response. It is common to see sets of values where the individual values make sense, but they do not fit together to make a coherent QAS.

Once a valid scenario is constructed, there is the question as to whether it is of any use. One property that reduces the usefulness of scenarios is when it assumes or dictates architectural decisions. It is common early on for students to, effectively, think about how they would build the system they are supposed to be developing QASs for, and then write the scenarios with their designs in mind. While it is possible that some aspects of an architecture may be dictated by the client ("must use J2EE" or even "must use replication to ensure availability") it is not appropriate for the architect to add architectural details not already given in the requirements.

Even once we have valid scenarios that really do specify requirements, there is still the question as to whether or not they specify the right requirements, that is, those intended by the client. While this is a crucial property of a useful scenario — if we get it wrong then the wrong architecture may result — it is in some sense the one I'm least concerned about. If the students can produce scenarios that a client can immediately determine are specifying the wrong things, then I have done my job. If the QASs have been produced to a level of quality that clients can, with confidence, figure out that they are the wrong requirements, then at least the (big) problem of mis-communication has been reduced.

5.2 Examples

I will now illustrate the comments above with examples.

In the first QAS exercise, the students are asked to produce performance scenarios for the Bus Tracker system. As well as producing the concrete scenario, the students are also required to give the general scenario (ideally starting with that), and also explain why they have picked the scenario they have in terms of the details of the system they have been given. The latter requirement is intended to prevent students choosing a scenario they think *should* exist, as opposed to one based on the information provided by the client. This is to remind students that once they become professional engineers, their responsibility is to their client, and so they are not free to just make up stuff they think might be interesting. That said, as I mentioned earlier, I don't provide complete details and so they do sometimes have to fill in the gaps. My main requirement is that whatever assumptions they make are not inconsistent (note the deliberate use of the double negative) with the text I give.

Figure 4 shows a typical first attempt at a performance QAS for the Bus Tracker system (in fact produced by one of the teams in the 2007 class). The first point to make about this scenario is that the team has identified a reasonable requirement for the Bus-Tracker system, namely that providing the estimated arrival times on the displays in a timely manner is a key requirement. The second point to make is that the general scenario details are reasonable for the requirement they are trying to specify. This suggests that any problems the students have is not due to lack of understanding as to what they are trying to do, or the general idea of what a scenario looks like. The problems are with their choice of values for the concrete scenario, both for the specific parts, and for the overall scenario.

The first problem is with the stated stimulus: "a bus is within one hour of arrival from the specific location." This does not describe something that might be regarded as an actual stimulus. My experience is that the wording is indicative of confusion as to what constitutes a stimulus, and this is an issue that needs further discussion and explanation.

A generous interpretation of what has been written might assume that it was intended to be something like "a bus reaches a point that is one hour from the specific location," which is closer to being a stimulus, however it is a somewhat nebulous statement, and, more significantly is inconsistent with both the stated source "gps transmitter on the bus" and the stated artifact "bustrack system" — the transmitter doesn't cause a bus to arrive anywhere, and the bus' arrival at some point doesn't directly have any effect on the overall system. Note that in this case it is not the values of source and artifact that are the problem, but the relationships between values for different parts of the scenario.

A stimulus value that would be more consistent with the stated source and artifact would be something like "a message from a bus within one hour from the location is received." This suggests that there needs to be another scenario for when the buses are further away. As QAS are intended to capture quality requirements, there should only be separate scenarios if the quality requirements are different. In this case, it might be reasonable to suppose that the display does not have to be updated in quite so timely a manner.

The choice of source indicates some confusion. For a start, a GPS system typically doesn't transmit anything, but that could be due to either lack of knowledge about how these things work, or what was written being shorthand for "The subsystem on the bus containing the GPS receiver." Nevertheless it is representative of a lack of precision regarding what exactly constitutes the values of the different parts of the scenarios.

The choice of artifact also shows a lack of precision. In the context given, we can probably reasonably assume that it is the "central" system that is meant, but, as with any presentation of requirements, we would prefer not to have to assume anything.

While beginners generally seem to understand what most of the parts are supposed to do (even if they struggle to apply this understanding to produce sensible values), the environment part usually causes the most confusion in terms of its purpose. The value given in the example "Less than 5 buses on the display" is, on the face of it, not an un-

reasonable value. However it implies that there must be a different scenario for the case when there are more than 5 buses (needed to be) on the display. While it is likely that in terms of *implementation* these two cases may be different, it seems unlikely that the quality requirements for the two cases should be different, and so separate scenarios are unnecessary. I believe this confusion is due to the students still thinking in terms of what they have to do to produce the system, rather than thinking about what the quality requirements of the system are. The fact that scenarios are intended to specify *requirements* (not design or implementation details) and in particular *quality* requirements is in my experience something that needs to be repeated and reinforced.

One point to make about this example is, once the issues regarding the other parts of the scenario are resolved, the stated measure, "The display is updated within 15 seconds," and response, "The Bus is displayed on the sign," are reasonable values. It is not uncommon to get measures such as "before the bus gets to the next stop", which, as well as not really being a measurement, is not something that any proposed architecture could be evaluated against.

Once all the issues discussed above are resolved, there is still the question as to how useful the resulting scenario is. It may describe something that looks like a performance requirement, but is it one that we would care about when developing an architecture? What this scenario does not do is specify which display is being updated. A bus does not go to just one bus stop, and most of the time it will be within one hour of several bus stops, with more than one having displays. It is not sufficient that the system update on bus stop in a timely manner, but that it update several. The need to generate multiple estimates and deliver them to multiple displays is likely to be significant in terms of performance requirements. This is a point that is always missed.

Related to the above point is the fact that there are also multiple buses, including multiple buses on the same route (and so visiting the same displays in the same order). This point is not directly evident in the given scenario. However, it seems clear that the team was somewhat aware of this due to their choice of "sporadic event" for the general value of the scenario, which only makes sense if there are multiple (albeit unpredictably spaced) events.

There is one remaining problem with the example scenario, which I will address in section 6.

5.3 Other comments

As I mentioned earlier there was usually little difficulty in coming up with a consistent general scenario, but there are problems deciding on concrete values to match the general values. In part, some of those problems are due confusion as to what the general values meant. For example, distinction between "sporadic" and "stochastic" was often unclear, as are the differences between "latency", "throughput", and "deadline." Even when it was clear that these terms had been encountered before, there appeared to be difficulty applying them in this context. This showed up in performance discussions most often, but that is almost certainly because that was the quality attribute I used the most, figuring students would have better intuition about it than the less familiar quality attributes.

6 Bus Tracker Scenarios

In this section I will present some of the scenarios I use in the course to illustrate various points about QAS development.

The starting point is to reexamine the information that has been provided. While I make the point that we have a responsibility to the client, that doesn't mean we should simply accept what we are given uncritically! Close examination of the given text reveals some problems. For example, consider the phrase "*the estimated arrival time should be within 2 minutes of the actual time, 95% of the time*." While it seems reasonable that the estimate should be reasonably accurate (otherwise what is the point of having it), it is not clear what "95% of the time" means. If this is determined over a day's operation, which is from 6am to 12am, then if the system is down for 1 hour it will not meet this requirement. However, 1 hour of downtime over a week does meet this requirement.

Another problem is the phrase "*Once the bus is within 1 kilometre*" which is intended to indicate that if the bus is a reasonable distance away then the estimates can be less accurate. However, it takes 3 minutes to travel 1 kilometre at 20 km/h. Having a 2 minute accuracy requirement when the bus is only about that far away seems fairly pointless! Notice that these issues didn't cause the problems the students faced in developing their scenarios.

Instead, we need to determine what the client is really trying to say. One reasonable interpretation is:

> *Provide estimated times of arrival accurate enough to be useful.*

Now we can start considering the quality implications of this requirement. Examples include:

- can the system "keep up," that is show estimates in time to be useful? — *performance*
- can we add new displays quickly and/or cheaply while still showing accurate enough estimates in time to be useful? — *modifiability*
- under what conditions must the system show accurate enough estimates in time to be useful? — *availability*

Starting with performance, we need to express what it means for the system to "keep up" or, "show accurate enough estimated times of arrivals for all buses on all displays". A question that is always raised at this point is concern of how to specify this requirement without knowing how to get accurate enough estimates, or whether the estimates can be produced fast enough. There are two answers I give to this question. The first is that, while as engineers we are responsible for building the system, that does not mean we have to build every little bit of it ourselves. For such things as estimation algorithms, we could contract that out to experts (e.g., those with a more traditional computer science background). The second answer is that that question isn't actually relevant when trying to come up with QASs. The client determines how accurate the estimate has to be and that dictates how quickly the estimates have to be generated. It is up to us, once we know what "quickly" means, to find develop an architecture that will meet the requirements (or convince the client to accept something a little less expensive). For the moment, the question that needs to be answered is what "quickly" means.

Source	A bus subsystem
Stimulus	. . . sends out a message with its current speed and location every 15 seconds
Artifact	. . . to the central subsystem
Environment	. . . when all communications and hardware is working adequately.
Response	The system produces an estimated arrival time for all relevant displays and sends them out to the display where the estimate is shown
Measure	. . . in under 30 seconds from when the bus' message was sent.

Figure 5: Performance Scenario A

Figure 5 gives my version of a scenario like that shown in the previous section. It has an issue that the students are

quick to point out — how does it make sense to have 15 seconds in one place and 30 in an other? Note that this is not a problem with the integrity of the scenario itself, but is a perceived problem with the requirements it is supposed to capture. This is what is good about QASs. They allow for a discussion about what the precise requirements are, rather than having to guess.

Source	A bus subsystem
Stimulus	. . . sends out a message with its current speed and location every **60** seconds
Artifact	. . . to the central subsystem
Environment	. . . when all communications and hardware is working adequately.
Response	The system produces an estimated arrival time for all relevant displays and sends them out to the display where the estimate is shown
Measure	. . . in under **15** seconds from when the bus' message was sent.

Figure 6: Performance Scenario B

Figure 6 shows a scenario that fixes the problem with Scenario A. It says that each bus sends out its speed and location every 60 seconds, and all updated estimates based on that information must be shown within 15 seconds of the message being sent. This is, however, a more subtle problem with this scenario — it suggests that a new estimate be delivered to every relevant display every time every bus sends a message. In fact, there is nothing in the requirements implied by this scenario that prevents an implementation just sending an old estimate (if it's not too old). Nevertheless the scenario does seem to be specifying more than is intended.

Source	A bus subsystem
Stimulus	. . . sends out a message with its current speed and location every 5 seconds
Artifact	. . . to the central subsystem
Environment	. . . when all communications and hardware is working adequately.
Response	The system stores the bus' id, location, and speed with a timestamp for when the message was received
Measure	. . . in under 1 second from when the message is received.

Figure 7: Performance Scenario C

Figure 7 shows a better scenario, in that all it specifies is that messages that are sent by buses are quickly recorded. But a feature of all of these scenarios is that they are "internal", in that the stimuli are all generated from within the system, and they imply part of the architecture (the three subsystems and to some extent how they interact). The latter point should especially be of concern. For example, all three scenarios assume that the bus subsystems "push" their information to the rest of the system, rather than the central system polling buses for their current speed and location. While the push style interaction probably makes most sense, requirements that imply design are to be avoided.

Figure 8 shows a scenario that I feel is more useful for describing some of the performance requirements for Bus Tracker. The stimulus is external to the system, namely a user of the system wanting to use its functionality, and all it says is that what is shown on the display should be based on fairly current information about the relevant bus' speed and location. Note the level of detail provided in the environment part. This performance requirement does not apply if there are more than 20 buses scheduled to arrive (so, so long as the display can scroll through 20 buses in 30 seconds the patron will see her bus' time within 30 seconds), nor does it apply if the bus is estimated to be more

Source	A bus patron
Stimulus	. . . wants to know when their bus is going to arrive at
Artifact	. . . the bus stop they're standing at when
Environment	. . . the desired bus is estimated to arrive in less than 20 minutes and at most 20 buses are estimated to arrive within 20 minutes of the bus stop.
Response	The display at the bus top shows an estimated time of arrival of the desired bus based on the actual location of the bus that is not more than 1 minute old
Measure	. . . in under 30 seconds.

Figure 8: Performance Scenario D

than 20 minutes away (although presumably the estimated time will *eventually* appear on the display, just not in 30 seconds — another scenario would be needed to specify that).

Now consider the modifiability requirement. Adding a new display can be a non-trivial operation. The equipment needs to be assembled, a hole dug and wires connected. If we were responsible for building the entire system, we might need to make decisions regarding this aspect, but in terms of developing a *software* architecture, it's the software modifications we need to consider. The kinds of questions we need to think about includes such things as: does the change have to be possible when the system is operational or can we shut the system down, how do we measure the cost of modification — time, money, or something else.

Source	Operations manager
Stimulus	. . . requests that a new display be made operational
Artifact	. . . in the Bus Tracker system
Environment	. . . outside of the peak traffic period.
Response	The new display starts displaying estimated arrival times for buses relevant to it
Measure	. . . within 5 minutes of the request

Figure 9: Modifiability Scenario E

Figure 9 shows the scenario that requires that a new display must be able to be brought on-line without affecting the rest of the system, but only during a time that is outside peak traffic (perhaps because the client does not want to do anything that will affect the system's behaviour during the time when most of the patrons want to use it). This scenario assumes that all the equipment is already in place.

Source	Operations manager
Stimulus	. . . requests that a new display be made operational
Artifact	. . . in the Bus Tracker system
Environment	. . . **while no buses are running**
Response	The new display is available to start displaying estimated arrival times for buses relevant to it
Measure	. . . **within 4 hours**.

Figure 10: Modifiability Scenario F

Figure 10 shows another possibility. If the client agreed that this scenario was acceptable then it would allow the system to be shut down and restarted, which may allow a choice of architecture (and so system) that is cheaper than what would be required to meet Scenario E.

For the availability requirements, we need to consider what constitutes "useful", for example this might mean

Source	A random event
Stimulus	. . . causes a failure
Artifact	. . . to the subsystem on a bus
Environment	. . . while the bus is on a bus route.
Response	The relevant displays must start showing the scheduled arrival time for the bus
Measure	. . . until it next starts a route

Figure 11: Availability Scenario G

"most" of the time estimated arrival times for any given bus must be shown, and the remainder of the time it's acceptable to show the scheduled arrival time. Figures 11 and 12 show two possibilities. Scenario G represents the expectation that the failure of a single bus' subsystem should not impact the rest of the system and be fixable once the bus returns to base (that is, fast enough that it can be done before the next time the bus goes out on a route). Note that this measure for this scenario is not really a useful one as presented. We really need specific details as to how long bus routes are and how quickly buses get turned around to go out on another route.

Source	A random event
Stimulus	. . . causes a failure
Artifact	. . . to the communications network
Environment	. . . during normal operation.
Response	All displays must start showing the scheduled arrival time for all buses
Measure	. . . within 30 seconds of the failure.

Figure 12: Availability Scenario H

Scenario H indicates what has to happen when the communications network goes down. One possibility could have been to just show nothing in this case, but this scenario requires that all the displays showed the scheduled arrival times for all buses. This requirement has an architectural implication. It implies that all displays must have access to these scheduled times, which means that they will need local copies of those times.

7 Concluding Comments

I have described some of the issues I have encountered in teaching the concept of Quality Attribute Scenarios to third year software engineering students. I have taught software architecture for 6 years now, 5 of which using QAS. I have presented some of the details of the Bus Tracker case study to aid my teaching of this concept.

My goal in teaching software architecture is not to produce software architects — as with any other engineering discipline the theory has to be tempered with real-world experience before it becomes useful. Many have suggested that to teach software architecture (and software engineering in general) the coursework needs to be as "real world" as possible. This is, however, difficult in a university environment, and I think we must be content with giving as much practical application of the theory as possible. My conclusion based on my experiences teaching this course is that, at least at the level I am teaching, the students don't have the necessary experience to appreciate a "real world" example. They are still struggling to understand the fundamental principles, and if these are not properly understood, no amount of real world examples will help.

In the case of QASs, practical application takes the form of multiple cycles of creating scenarios and evaluating them for multiple applications. My goal is to make sure that the theory is understood. Based on the assessment items relating to QASs (assignment and typically an exam question) I am meeting that goal, at least with respect to the QAS concept.

Acknowledgements

My thanks to the students I have taught in the 6 years of software architecture teaching I have done. They asked the hard questions that exposed the issues. In particular, thanks to the 2007 Part III software engineering cohort, whose examples have been used here with their permission.

References

Bass, L., Clements, P. & Kazman, R. (2003), *Software Architecture in Practice*, 2 edn, Addison-Wesley.

Bucci, P., Long, T. J. & Weide, B. W. (1998), Teaching software architecture principles in CS1/CS2, *in* 'ISAW '98: Proceedings of the third International Workshop on Software Architecture', ACM Press, New York, NY, USA, pp. 9–12.

Garlan, D., Allen, R. & Ockerbloom, J. (1995), 'Architectural mismatch: Why reuse is so hard', *IEEE Software* **12**(6), 17–26.

Gruba, P., Moffat, A., Sondergaard, H. & Zobel, J. (2004), What drives curriculum change?, *in* R. Lister & A. L. Young, eds, 'Sixth Australasian Computing Education Conference (ACE2004)', Vol. 30 of *CRPIT*, ACS, Dunedin, New Zealand, pp. 109–117.

Hamer, J., Kell, C. & Spence, F. (2007), Peer assessment using Aropä, *in* 'ACE '07: Proceedings of the ninth Australasian conference on Computing education', Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 43–54.

Karam, O., Qian, K. & Diaz-Herrera, J. (2004), A model for SWE course "software architecture and design", *in* '34th Annual Frontiers in Education (FIE)', pp. 4–8.

Lago, P. & van Vliet, H. (2005), Teaching a course on software architecture, *in* '18th Conference on Software Engineering Education and Training (CSEE&T)', pp. 35–42.

Royce, W., Boehm, B. & Druffel, C. (1994), Employing unas technology for software architecture education at the university of southern california, *in* 'WADAS '94: Proceedings of the eleventh annual Washington Ada symposium & summer ACM SIGAda meeting on Ada', ACM Press, New York, NY, USA, pp. 113–121.

SEI (2007), 'SEI list of definitions of software architecture', `http://www.sei.cmu.edu/architecture/definitions.html` Accessed September.

Shaw, M. & Clements, P. (2006), 'The golden age of software architecture', *IEEE Software* **23**(2), 31–39.

Shaw, M. & Garlan, D. (1996), *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.