19

Architecturally Significant Requirements

> *The most important single aspect of software development is to be clear about what you are trying to build.*
>
> —Bjarne Stroustrup, creator of C++

Architectures exist to build systems that satisfy requirements. By "requirements," we do not necessarily mean a documented catalog produced using the best techniques that requirements engineering has to offer. Instead, we mean the set of properties that, if not satisfied by your system, will cause the system to be a failure. Requirements exist in as many forms as there are software development projects—from polished specifications to verbal shared understanding (real or imagined) among principal stakeholders. The technical, economic, and philosophical justifications for your project's requirements practices are beyond the scope of this book. What is in scope is that, regardless of how they are captured, they establish the criteria for success or failure, and architects need to know them.

To an architect, not all requirements are created equal. Some have a much more profound effect on the architecture than others. An *architecturally significant requirement (ASR)* is a requirement that will have a profound effect on the architecture—that is, the architecture might well be dramatically different in the absence of such a requirement.

You cannot hope to design a successful architecture if you do not know the ASRs. ASRs often, but not always, take the form of quality attribute (QA) requirements—the performance, security, modifiability, availability, usability, and so forth, that the architecture must provide to the system. In Chapters 4–14, we introduced patterns and tactics to achieve QAs. Each time you select a pattern or tactic to use in your architecture, you are do-

ing so because of the need to meet QA requirements. The more difficult and important the QA requirement, the more likely it is to significantly affect the architecture, and hence to be an ASR.

Architects must identify ASRs, usually after doing a significant bit of work to uncover candidate ASRs. Competent architects know this. Indeed, as we observe experienced architects going about their duties, we notice that the first thing they do is start talking to the important stakeholders. They're gathering the information they need to produce the architecture that will respond to the project's needs—whether or not this information has been previously identified.

This chapter provides some systematic techniques for identifying the ASRs and other factors that will shape the architecture.

19.1 Gathering ASRs from Requirements Documents

An obvious location to look for candidate ASRs is in the requirements document or in user stories. After all, we are looking for requirements, and requirements should be (duh) in requirements documents. Unfortunately, this is not usually the case, although information in the requirements documents can certainly be useful.

Don't Get Your Hopes Up

Many projects don't create or maintain the kind of requirements document that professors in software engineering classes or authors of traditional software engineering books love to prescribe. Furthermore, no architect just sits and waits until the requirements are "finished" before starting work. The architect *must* begin while the requirements are still in flux. Consequently, the QA requirements are quite likely to be uncertain when the architect starts work. Even where they exist and are stable, requirements documents often fail an architect in two ways:

- Most of the information found in a requirements specification does not affect the architecture. As we've seen over and over, architectures are mostly driven or "shaped" by QA requirements, which determine

and constrain the most important architectural decisions. Even so, the vast bulk of most requirements specifications focus on the required features and functionality of a system, which shape the architecture the least. The best software engineering practices do prescribe capturing QA requirements. For example, the Software Engineering Body of Knowledge (SWEBOK) says that QA requirements are like any other requirements: They must be captured if they are important, and they should be specified unambiguously and be testable.

In practice, though, we rarely see adequate capture of QA requirements. How many times have you seen a requirement of the form "The system shall be modular" or "The system shall exhibit high usability" or "The system shall meet users' performance expectations"? These are not useful requirements because they are not testable; they are not falsifiable. But, looking on the bright side, they can be viewed as invitations for the architect to begin a conversation about what the requirements in these areas really are.

- Much of what is useful to an architect won't be found in even the best requirements document. Many concerns that drive an architecture do not manifest themselves at all as observables in the system being specified, and thus are not the subject of requirements specifications. ASRs often derive from business goals in the development organization itself; we'll explore this connection in Section 19.3. Developmental qualities are also out of scope; you will rarely see a requirements document that describes teaming assumptions, for example. In an acquisition context, the requirements document represents the interests of the acquirer, not those of the developer. Stakeholders, the technical environment, and the organization itself all play a role in influencing architectures. When we discuss architecture design, in Chapter 20, we will explore these requirements in more detail.

Sniffing out ASRs from a Requirements Document

While requirements documents won't tell an architect the whole story, they are still an important source of ASRs. Of course, ASRs will not be conveniently labeled as such; the architect should expect to perform a bit of investigation and archaeology to ferret them out.

Some specific things to look for are the following categories of information:

- *Usage*. User roles versus system modes, internationalization, language distinctions.
- *Time*. Timeliness and element coordination.
- *External elements*. External systems, protocols, sensors or actuators (devices), middleware.
- *Networking*. Network properties and configurations (including their security properties).
- *Orchestration*. Processing steps, information flows.
- *Security properties*. User roles, permissions, authentication.
- *Data*. Persistence and currency.
- *Resources*. Time, concurrency, memory footprint, scheduling, multiple users, multiple activities, devices, energy usage, soft resources (e.g., buffers, queues), and scalability requirements.
- *Project management*. Plans for teaming, skill sets, training, team coordination.
- *Hardware choices*. Processors, families of processors, evolution of processors.
- *Flexibility of functionality, portability, calibrations, configurations.*
- *Named technologies, commercial packages.*

Anything that is known about their planned or anticipated evolution will be useful information, too.

Not only are these categories architecturally significant in their own right, but the possible change and evolution of each are also likely to be architecturally significant. Even if the requirements document you're mining doesn't mention evolution, consider which of the items in the preceding list are likely to change over time, and design the system accordingly.

19.2 Gathering ASRs by Interviewing Stakeholders

Suppose your project isn't producing a comprehensive requirements document. Or maybe it is, but it won't have the QAs nailed down by the time

you need to start your design work. What do you do?

First, stakeholders often don't know what their QA requirements actually are. In that case, architects are called upon to help set the QA requirements for a system. Projects that recognize this need for collaboration and encourage it are much more likely to be successful than those that don't. Relish the opportunity! No amount of nagging your stakeholders will suddenly instill in them the necessary insights. If you insist on quantitative QA requirements, you may get numbers that are arbitrary and at least some of those requirements will be difficult to satisfy and, in the end, actually detract from system success.

Experienced architects often have deep insights into which QA responses have been exhibited by similar systems, and which QA responses are reasonable to expect and to provide in the current context. Architects can also usually give quick feedback as to which QA responses will be straightforward to achieve and which will likely be problematic or even prohibitive.

For example, a stakeholder may ask for 24/7 availability—who wouldn't want that? However, the architect can explain how much that requirement is likely to cost, which will give the stakeholders information to make a tradeoff between availability and affordability. Also, architects are the only people in the conversation who can say, "I can actually deliver an architecture that will do better than what you had in mind—would that be useful to you?"

Interviewing the relevant stakeholders is the surest way to learn what they know and need. Once again, it behooves a project to capture this critical information in a systematic, clear, and repeatable way. Gathering this information from stakeholders can be achieved by many methods. One such method is the Quality Attribute Workshop (QAW), described in the sidebar.

THE QUALITY ATTRIBUTE WORKSHOP

The QAW is a facilitated, stakeholder-focused method to generate, prioritize, and refine quality attribute scenarios before the software architecture is completed. It emphasizes system-level concerns and specifically the role that software will play in the system. The QAW is keenly dependent on the participation of system stakeholders.

After introductions and an overview of the workshop steps, the QAW involves the following elements:

- *Business/mission presentation.* The stakeholder representing the business concerns behind the system (typically a manager or management representative) spends about one hour presenting the system's business context, broad functional requirements, constraints, and known QA requirements. The QAs that will be refined in later steps will be derived largely from the business/mission needs presented in this step.
- *Architectural plan presentation.* While a detailed system or software architecture might not exist, it is possible that broad system descriptions, context drawings, or other artifacts have been created that describe some of the system's technical details. At this point in the workshop, the architect will present the system architectural plans as they stand. This lets stakeholders know the current architectural thinking, to the extent that it exists.
- *Identification of architectural drivers.* The facilitators will share their list of key architectural drivers that they assembled in the prior two steps, and ask the stakeholders for clarifications, additions, deletions, and corrections. The idea is to reach a consensus on a distilled list of architectural drivers that include overall requirements, business drivers, constraints, and quality attributes.
- *Scenario brainstorming.* Each stakeholder expresses a scenario representing his or her concerns with respect to the system. Facilitators ensure that each scenario addresses a QA concern, by specifying an explicit stimulus and response.
- *Scenario consolidation.* After the scenario brainstorming, similar scenarios are consolidated where reasonable. Facilitators ask stakeholders to identify those scenarios that are very similar in content.

Scenarios that are similar are merged, as long as the people who proposed them agree and feel that their scenarios will not be diluted in the process.

- *Scenario prioritization.* Prioritization of the scenarios is accomplished by allocating each stakeholder a number of votes equal to 30 percent of the total number of scenarios generated after consolidation. Stakeholders can allocate any number of their votes to any scenario or combination of scenarios. The votes are counted, and the scenarios are prioritized accordingly.
- *Scenario refinement.* After the prioritization, the top scenarios are refined and elaborated. Facilitators help the stakeholders put the scenarios in the six-part scenario form of source–stimulus–artifact–environment–response–response measure that we described in Chapter 3. As the scenarios are refined, issues surrounding their satisfaction will emerge and should be recorded. This step lasts as long as time and resources allow.

The results of stakeholder interviews should include a list of architectural drivers and a set of QA scenarios that the stakeholders (as a group) prioritized. This information can be used for the following purposes:

- Refine system and software requirements.
- Understand and clarify the system's architectural drivers.
- Provide a rationale for why the architect subsequently made certain design decisions.
- Guide the development of prototypes and simulations.
- Influence the order in which the architecture is developed.

I DON'T KNOW WHAT THAT REQUIREMENT SHOULD BE

It is not uncommon when interviewing stakeholders and probing for ASRs that they will complain, "I don't know what that requirement should be." While it is true that this is the way that they *feel*, it is also frequently the case that they know *something* about the requirement, particularly if the stakeholders are experienced in the domain. In this case, elic-

iting this “something” is far better than simply making up the requirement on your own. For example, you might ask, “How quickly should the system respond to this transaction request?” If the answer is “I don’t know,” my advice here is to play dumb. You can say, “So . . . 24 hours would be OK?” The response is often an indignant and astonished “No!” “Well, how about 1 hour? “No!” “Five minutes? “No!” “How about 10 seconds?” “Well, <grumble, mumble> I suppose I could live with something like that. . . .”

By playing dumb, you can often get people to at least give you a range of acceptable values, even if they do not know precisely what the requirement should be. And this range is typically enough for you to choose architectural mechanisms. A response time of 24 hours versus 10 minutes versus 10 seconds versus 100 milliseconds means, to an architect, choosing very different architectural approaches. Armed with this information, you can now make informed design decisions.

—RK

19.3 Gathering ASRs by Understanding the Business Goals

Business goals are the *raison d’être* for building a system. No organization builds a system without a reason; rather, the people involved want to further the mission and ambitions of their organization and themselves. Common business goals include making a profit, of course, but most organizations have many more concerns than simply profit. In still other organizations (e.g., nonprofits, charities, governments), profit is the furthest thing from anyone’s mind.

Business goals are of interest to architects because they frequently lead directly to ASRs. There are three possible relationships between business goals and an architecture:

1. *Business goals often lead to quality attribute requirements*. Every quality attribute requirement—such as user-visible response time or platform flexibility or iron-clad security or any of a dozen other needs—

originates from some higher purpose that can be described in terms of added value. A desire to differentiate a product from its competition and let the developing organization capture market share may lead to a requirement for what might seem like an unusually fast response time. Also, knowing the business goal behind a particularly stringent requirement enables the architect to question the requirement in a meaningful way—or marshal the resources to meet it.

2. *Business goals may affect the architecture without inducing a quality attribute requirement at all.* A software architect related to us that some years ago he delivered an early draft of the architecture to his manager. The manager remarked that a database was missing from the architecture. The architect, pleased that the manager had noticed, explained how he (the architect) had devised a design approach that obviated the need for a bulky, expensive database. The manager, however, pressed for the design to include a database, *because the organization had a database unit employing a number of highly paid technical staff who were currently unassigned and needed work*. No requirements specification would capture such a requirement, nor would any manager allow such a motivation to be captured. And yet that architecture, had it been delivered without a database, would have been just as deficient—from the manager's point of view—as if it had failed to deliver an important function or QA.
3. *No influence of a business goal on the architecture*. Not all business goals lead to quality attributes. For example, a business goal to "reduce cost" might be realized by lowering the facility's thermostats in the winter or reducing employees' salaries or pensions.

Figure 19.1 illustrates the major points from this discussion. In the figure, the arrows mean "leads to." The solid arrows highlight the relationships of greatest interest to architects.

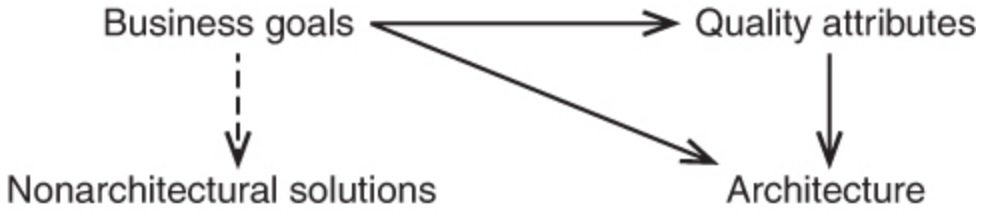

Figure 19.1 Some business goals may lead to quality attribute requirements, or lead directly to ar-

chitectural decisions, or lead to non-architectural solutions.

Architects often become aware of an organization's business and business goals via osmosis—working, listening, talking, and soaking up the goals that are at work in an organization. Osmosis is not without its benefits, but more systematic ways of determining such goals are both possible and desirable. Moreover, it is worthwhile to capture business goals explicitly, because they often imply ASRs that would otherwise go undetected until it is too late or too expensive to address them.

One way to do this is to employ the PALM method, which entails holding a workshop with the architect and key business stakeholders. The heart of PALM consists of these steps:

- *Business goals elicitation*. Using the categories given later in this section to guide the discussion, capture from stakeholders the set of important business goals for this system. Elaborate the business goals and express them as business goal scenarios.[1] Consolidate almost-alike business goals to eliminate duplication. Have the participants prioritize the resulting set to identify the most important goals.

1. A business goal scenario is a structured seven-part expression that captures a business goal, similar in intent and usage to a QA scenario. This chapter's "For Further Reading" section contains a reference that describes PALM, and business goal scenarios, in full detail.

- *Identify potential QAs from business goals*. For each important business goal scenario, have the participants describe a QA and response measure value that (if architected into the system) would help achieve the goal.

The process of capturing business goals is well served by having a set of candidate business goals handy to use as conversation-starters. If you know that many businesses want to gain market share, for instance, you can use that motivation to engage the right stakeholders in your organization: "What are our ambitions about market share for this product, and how could the architecture contribute to meeting them?"

Our research in business goals has led us to adopt the categories shown in the list that follows. These categories can be used as an aid to brainstorming and elicitation. By employing the list of categories, and asking the stakeholders about possible business goals in each category, some assurance of coverage is gained.

1. Growth and continuity of the organization
2. Meeting financial objectives
3. Meeting personal objectives
4. Meeting responsibility to the employees
5. Meeting responsibility to society
6. Meeting responsibility to the state
7. Meeting responsibility to the shareholders
8. Managing market position
9. Improving business processes
10. Managing the quality and reputation of products
11. Managing change in the environment over time

19.4 Capturing ASRs in a Utility Tree

In a perfect world, the techniques described in Sections 19.2 and 19.3 would be applied early on in your development process: You would interview the key stakeholders, elicit their business goals and driving architectural requirements, and have them prioritize all of these inputs for you. Of course, the real world, lamentably, is less than perfect. It is often the case that you do not have access to these stakeholders when you need them, for organizational or business reasons. So what do you do?

Architects can use a construct called a *utility tree* when the "primary sources" of requirements are not available. A utility tree is a top-down

representation of what you, as an architect, believe to be the QA-related ASRs that are critical to the success of the system.

A utility tree begins with the word “Utility” as the root node. Utility is an expression of the overall “goodness” of the system. You then elaborate on this root node by listing the major QAs that the system is required to exhibit. (You might recall that we said in Chapter 3 that QA names by themselves were not very useful. Never fear—they are only being used as intermediate placeholders for subsequent elaboration and refinement!)

Under each QA, record specific refinements of that QA. For example, performance might be decomposed into “data latency” and “transaction throughput” or, alternatively, “user wait time” and “time to refresh web page.” The refinements that you choose should be the ones that are relevant to your system. Under each refinement, you can then record the specific ASRs, expressed as QA scenarios.

Once the ASRs are recorded as scenarios and placed at the leaves of the tree, you can evaluate these scenarios against two criteria: the business value of the candidate scenario and the technical risk of achieving it. You can use any scale you like, but we find that a simple “H” (high), “M” (medium), and “L” (low) scoring system suffices for each criterion. For business value, “high” designates a must-have requirement, “medium” identifies a requirement that is important but would not lead to project failure were it omitted, and “low” describes a nice requirement to meet but not something worth much effort. For technical risk, “high” means that meeting this ASR is keeping you awake at night, “medium” means meeting this ASR is concerning but does not carry a high risk, and “low” means that you have confidence in your ability to meet this ASR.

Table 19.1 shows a portion of an example utility tree. Each ASR is labeled with an indicator of its business value and its technical risk.

Table 19.1 Tabular Form of the Utility Tree for a System in the Healthcare Space

Quality Attribute	Attribute Refinement	ASR Scenario
Performance	Transaction response time	A user updates a patient's account in response to a change-of-address notification while the system is under peak load, and the transaction completes in less than 0.75 seconds. (H, H)
	Throughput	At peak load, the system is able to complete 150 normalized transactions per second. (M, M)
Usability	Proficiency training	A new hire with two or more years' experience in the business can learn, with 1 week of training, to execute any of the system's core functions in less than 5 seconds. (M, L)
	Efficiency of operations	A hospital payment officer initiates a payment plan for a patient while interacting with that patient and completes the process with no input errors. (M, M)
Configurability	Data configurability	A hospital increases the fee for a particular service. The configuration team makes and tests the change in 1

Quality Attribute	Attribute Refinement	ASR Scenario
		working day; no source code needs to change. (H, L)
Maintainability	Routine changes	A maintainer encounters response-time deficiencies, fixes the bug, and distributes the bug fix with no more than 3 person-days of effort. (H, M)
		A reporting requirement requires a change to the report-generating metadata. Change is made and tested in 4 person-hours of effort (M, L)
	Upgrades to commercial components	The database vendor releases a new major version that is successfully tested and installed in less than 3 person-weeks. (H, M)
	Adding new feature	A feature that tracks blood bank donors is created and successfully integrated within 2 person-months. (M, M)

Quality Attribute	Attribute Refinement	ASR Scenario
Security	Confidentiality	A physical therapist is allowed to see that part of a patient's record dealing with orthopedic treatment, but not other parts or any financial information. (H, M)
	Resisting attacks	The system repels an unauthorized intrusion attempt and reports the attempt to authorities within 90 seconds. (H, M)
Availability	No down time	The database vendor releases new software, which is hot-swapped into place, with no downtime. (H, L)
		The system supports 24/7/365 web-based account access by patients. (M, M)

Once you have a utility tree filled out, you can use it to make important checks. For instance:

- A QA or QA refinement without any ASR scenario is not necessarily an error or omission that needs to be rectified, but rather an indication you should investigate whether there are unrecorded ASR scenarios in that area.
- ASR scenarios that receive a (H, H) rating are obviously the ones that deserve the most attention from you; these are the most significant of the significant requirements. A very large number of these scenarios

might be a cause for concern regarding whether the system is, in fact, achievable.

19.5 Change Happens

Edward Berard said, “Walking on water and developing software from a specification are both easy if both are frozen.” Nothing in this chapter should be taken to assume that such a miraculous state of affairs is likely to exist. Requirements—whether captured or not—change all the time. Architects have to adapt and keep up, to ensure that their architectures are still the right ones that will bring success to the project. In Chapter 25, where we discuss architecture competence, we’ll advise that architects need to be great communicators, and this means great bidirectional communicators, taking in as well as supplying information. Always keep a channel open to the key stakeholders who determine the ASRs so you can keep up with changing requirements. The methods offered in this chapter can be applied repetitively to accommodate change.

Even better than keeping up with change is staying one step ahead of it. If you get wind of a change to the ASRs, you can take preliminary steps to design for it, as an exercise to understand the implications. If the change will be prohibitively expensive, sharing that information with the stakeholders will be a valuable contribution, and the earlier they know it, the better. Even more valuable might be suggestions about changes that would do (almost) as well in meeting the goals but without breaking the budget.

19.6 Summary

Architectures are driven by architecturally significant requirements. An ASR must have:

- A profound impact on the architecture. Including this requirement will likely result in a different architecture than if it were not included.

- A high business or mission value. If the architecture is going to satisfy this requirement—potentially at the expense of not satisfying others—it must be of high value to important stakeholders.

ASRs can be extracted from a requirements document, captured from stakeholders during a workshop (e.g., a QAW), captured from the architect in a utility tree, or derived from business goals. It is helpful to record them in one place so that the list can be reviewed, referenced, used to justify design decisions, and revisited over time or in the case of major system changes.

In gathering these requirements, you should be mindful of the organization's business goals. Business goals can be expressed in a common, structured form and represented as business goal scenarios. Such goals may be elicited and documented using PALM, a structured facilitation method.

A useful representation of QA requirements is a utility tree. Such a graphical depiction helps to capture these requirements in a structured form, starting from coarse, abstract notions of QAs and gradually refining them to the point where they are captured as scenarios. These scenarios are then prioritized, with this prioritized set defining your "marching orders" as an architect.

19.7 For Further Reading

The Open Group Architecture Framework, available at opengroup.org/togaf/, provides a complete template for documenting a business scenario that contains a wealth of useful information. Although we believe architects can make use of a lighter-weight means to capture a business goal, it's worth a look.

The definitive reference source for the Quality Attribute Workshop is [Barbacci 03].

The term *architecturally significant requirement* was created by the SARA group (Software Architecture Review and Assessment), as part of a

document that can be retrieved at http://pkruchten.wordpress.com/architecture/SARAv1.pdf.

The Software Engineering Body of Knowledge (SWEBOK), third edition, can be downloaded here: computer.org/education/bodies-of-knowledge/software-engineering/v3. As we go to press, a fourth edition is being developed.

A full description of PALM [Clements 10b] can be found here: https://resources.sei.cmu.edu/asset_files/TechnicalNote/2010_004_001_15179.pdf.

19.8 Discussion Questions

1. Interview representative stakeholders for a business system in use at your company or your university and capture at least three business goals for it. To do so, use PALM's seven-part business goal scenario outline, referenced in the "For Further Reading" section.

2. Based on the business goals you uncovered for question 1, propose a set of corresponding ASRs.

3. Create a utility tree for an ATM. (Interview some of your friends and colleagues if you would like to have them contribute QA considerations and scenarios.) Consider a minimum of four different QAs. Ensure that the scenarios that you create at the leaf nodes have explicit responses and response measures.

4. Find a software requirements specification that you consider to be of high quality. Using colored pens (real ones if the document is printed; virtual ones if the document is online), color red all the material that you find completely irrelevant to a software architecture for that system. Color yellow all of the material that you think *might* be relevant, but not without further discussion and elaboration. Color green all of the material that you are certain is architecturally significant. When you're done, every part of the document that's not white space should be red, yellow, or green. Approximately what percentage of each color did your document end up being? Do the results surprise you?