CHAPTER 4

Empathize with Stakeholders

Knowing what problem to solve is sometimes easier said than done. Since we create software to help people, we must understand the people whose lives will be affected by the software we make to understand the problem thoroughly. The better we empathize with their needs, the better we'll see and understand the real problems that need to be solved.

We call people with an interest or concern in our software *stakeholders*. It's the architect's job to identify stakeholders and understand their needs. Our stakeholders' expectations for the system will directly or indirectly influence how we design it.

Empathy is the engine that drives design. When you empathize with the people affected by your software, then you'll make better software. In this chapter, you'll learn how to decide who to talk to about the problem you're solving and what you need to learn from them to start designing the architecture.

Talk to the Right People

Stakeholders usually, but not always, have a business interest in the software. They might pay for the software or directly profit from it. Users are important stakeholders but so too are the people who build and maintain the system. Other people might not even realize how our software might affect them, but it's sometimes necessary to consider their concerns as well.

In the wild, stakeholders rarely travel alone. We use the term *stakeholder group* to highlight this fact. Working with groups is different than working with individuals. Two people from the same stakeholder group can provide inconsistent or conflicting information. We must work to understand the whole group's concerns and sometimes even help them reach consensus.

Here are some stakeholders for the Project Lionheart Case Study, introduced on page 14. In this picture, the icons represent specific people or roles while speech bubbles show stakeholders' thoughts and feelings.

Stakeholders are interested in what we're building and will influence the architecture we design. Since we'll want to invite these people to future design workshops, we should find out who they are. Enter the stakeholder map.

Bett says:
It's About the Customer

By Bett Bollhoefer, software architect at General Electric

Architecture is about the customer. If I create an architecture that doesn't give value to the customer, I am wasting my time. When I talk to customers, I often hear horror stories about how their current systems were developed by someone in an ivory tower who didn't understand them or their work. How do I make sure I am bringing value to customers through my architecture?

My answer is to use a customer-centric design process. I start with who the customer is and what they want to do. I divide the system into tasks the customer performs. For each task, I find out how they start it and where they run into issues.

You might be thinking, "This doesn't sound like architecture—it sounds like user experience!" Yes, it is, but many UX designers don't understand the technical aspects of the system well enough to determine the architecture. My process goes beyond the

surface to ensure the deep structures support the customer's values. I call it *Customer Experience Architecture*.

Step 1: Determine what matters to the customer, including their functional requirements and quality attributes, by watching how they do the task in their natural environment and asking lots of *why's*.

Step 2: Design the system around the customer's needs and document it in a prototype. The prototype should be as interactive as possible, not just a flowchart.

Step 3: Review the prototype as early as possible with the customer. Make sure they really understand what is changing in the new system and how it will impact them.

Step 4: Revise the architecture based on feedback from the customer's review.

Using these four steps, you can create value for your customer through your architecture and become their hero, or at least not the person in the ivory tower who is ruining their life.

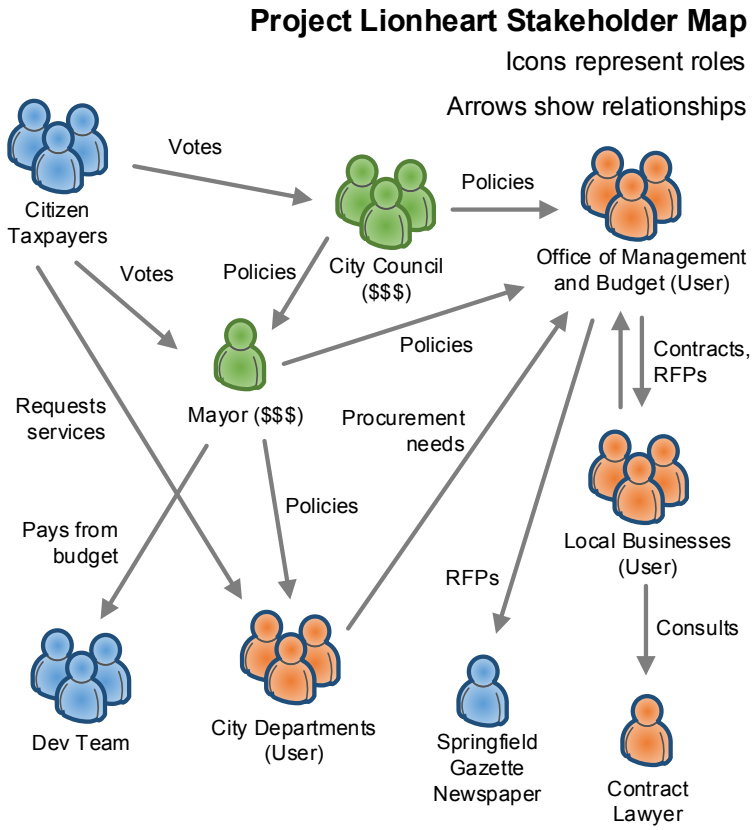
Create a Stakeholder Map

A *stakeholder map* is a network diagram showing all the people involved with or affected by the proposed software system. Stakeholder maps are ideal for visualizing relationships and interactions among people. They also give you a snapshot of what motivates different stakeholders. Use stakeholder maps to decide who the most important people are to talk to about their concerns.

Every time I create a stakeholder map, I'm surprised to see how many people I might touch with the software I build. There's a partial stakeholder map for Project Lionheart on page 42.

There are several stakeholders not shown on this diagram for the sake of simplicity. Additional stakeholders include IT vendors we might have to collaborate with, the Chamber of Commerce (or other lobbying organizations), the deputy mayor, and various community groups who receive services from the city. The *city departments* stakeholder can be made more precise by dividing it into the board of education, parks and recreation, public works, sanitation, and so on. If these groups have a similar stake in the system, then they could be lumped together as shown. As a rule of thumb, it's best to be as specific as possible.

Step back and look at the stakeholder map after you've created it. Who is paying for the software? Who is using it? Are there network hubs with many incoming or outgoing arrows? Are there stakeholders with potential conflicts of interest? These people are all excellent candidates for interviews and further research.



When I look at the Project Lionheart stakeholder map, I see a few interesting areas that we should investigate further.

1. Mayor van Damme hired us and we report to him, but the Office of Management and Budget receives policy direction from both the mayor and city council.
2. Our software will affect many city departments, but we won't be able to talk to all of them. We should identify a few representative stakeholders and carefully validate our findings with the larger group.
3. Some local businesses rely on lawyers to navigate the Request for Proposal process. Different interaction patterns for the potential software might exist, which could influence the architecture.
4. The Office of Management and Budget (OMB) sits at the center of several key user interactions, but they aren't paying the bills. We should talk to the OMB directly. It's possible that the mayor and city council have budgeted for a system that does not solve the OMB's real problems.

You can build a stakeholder map by yourself, but it's more fun to create them in groups. The steps for this activity are outlined in Activity 10, *Stakeholder Map*, on page 221.

Get Your Hands Dirty: Create a Stakeholder Map

Pick an open source project you use or contribute to and create a stakeholder map for it. Take a picture of it and share it on this book's forum.[1] Here are some things to think about:

- Is there an organization that oversees or funds the project? Are there subgroups within the organization who might have different vested interests?
- Who are the biggest contributors to the project?
- How is the project licensed? Who benefits from the choice of license?
- Who uses the project? What problems are they trying to solve?

Discover the Business Goals

Every software system is built to serve some fundamental purpose. *Business goals* describe what stakeholders hope to accomplish with the software. Business goals also seed conversations about quality attributes, trade-offs, and technical debt.

Business goals are a primary architectural driver and help prioritize competing concerns. The better everyone understands stakeholders' needs, the better you'll be able to help them. Here's a summary of common business goal categories:

Who wants it	What they want
Individuals	Increase wealth, power, reputation, personal enjoyment, or knowledge
Organizations	Increase revenue, maximize profits, grow the business, become a market leader, improve stability, enter a new market, beat a competitor
Employees	Interesting and meaningful work, increase knowledge, help users, become recognized as an expert
Development Team	Improve specific quality attributes, reduce costs, add new features, implement a standard, improve time-to-market
Nations, governments	Security, civic welfare, social responsibility, legal compliance

1. http://pragprog.com/book/mkdsa/design-it

Capture business goals as simple need-based statements that explain what stakeholders will get from the software system.

Record Business Goal Statements

Great business goal statements are measurable and have clear success criteria. Human-centered business goals allow your team to understand the people you are ultimately serving through the software you build.

Good business goal statements include three things:

Subject A specific person or role. If the stakeholder or group has a name, use it. *United Hamster Trainers Union* is better than *union groups*.

Outcome Express the stakeholder's need as a measurable outcome. How does the world change if the system is successful? You will design an architecture to achieve this outcome. For example, maybe the United Hamster Trainers Union needs a way to help members stay in touch with one another.

Context Context shares an insight about a stakeholder's need and helps build empathy. Ideally context is insightful and not completely obvious. For example, knowing that the United Hamster Trainers Union's most important annual meeting has over 5 million members attending virtually creates deeper understanding about the previously discussed outcome.

There are some business goals for Project Lionheart in the table on page 45. Putting business goal statements in a slick-looking table sometimes makes them easier to read.

Most systems only have only three to five business goals. More than this and the goals become confusing and difficult to remember. When working with many stakeholders, it's useful to record goals' relative importance. A simple *must have* or *nice to have* designation is good enough for this purpose.

Help Stakeholders Share Their Business Goals

Stakeholders usually know what they want, but many stakeholders find it difficult to articulate their needs as measurable statements. Every architect should have a few simple templates in their toolbox to help stakeholders find their voice. The point of view (POV) mad lib on page 45 is a fun alternative that is similar to a user story but describes the value expected from the whole system instead of functionality. Other business goal formats are described in Activity 8, *Point-of-View Mad Lib*, on page 215.

Stakeholder	Goal	Context
Mayor van Damme	Reduce procurement costs by 30%	Avoid making budget cuts to education or other essential services in an election year.
Mayor van Damme	Improve city engagement with local businesses, measured by the number of applications from first-time local businesses, percentage of overall RFPs won by local businesses.	Improve the local economy by ensuring local businesses win city contracts.
Office of Management and Business	Cut the time required to publish a new RFP in half.	Improves services across the city and reduces costs at the same time. Citizens suffer when city services go unfunded. Think: *No toilet paper at the girls' basketball game* or *not enough hypodermic needles for emergency medical crews.*
Office of Management and Business	Review historical procurement data for the past 10 years.	Businesses behave similarly over time and historic data gives the city a leg up when reviewing contract bids.

Mayor van Damme needs to reduce procurement costs by 30%
(stakeholder) *(stakeholder's need)*

because he wants to avoid cutting essential department funding.
(context)

Collaborate closely with your product manager or other business-focused stakeholders to identify the system's business goals. They can usually describe a system's business goals without breaking a sweat. Be prepared to help stakeholders or product managers if they struggle to articulate business goals, but remember they own the business goals.

 Get Your Hands Dirty: Create Business Goals for This System

What are the business goals for this proposed system? Create a few point of view mad libs for this scenario.

> *Bouncing Bean Grocery* is a regional grocery store chain. A few months ago an *Organic Plus!* store opened and *Bouncing Bean* has seen a decline in sales. Hoping to entice customers to their stores, *Bouncing Bean* has hired your team to develop a mobile application in which potential shoppers can create shopping lists, search recipes, and clip e-coupons. *Bouncing Bean* hopes the app will attract customers and provide customer data to drive targeted advertising.

Here are some things to think about.

- Who are the stakeholders? What do they hope to gain?
- Who are the users? What are they trying to accomplish? (Hint: It has nothing to do with software.)
- What's the worst that can happen? Sometimes thinking about failure can help uncover a business goal. People usually want to avoid failures.

Project Lionheart: The Story So Far…

Mayor van Damme gave our product manager a good starting point with his key strategic directives. The first thing she did was verify the mayor's business goals with other stakeholders. After looking at our stakeholder map (described on page 42), our product manager scheduled meetings with the head of the Office of Management and Budget and two members of city council.

Our product manager led the stakeholder interviews. You observed the interviews to develop a bit more empathy for our stakeholders' needs. Our product manager summarized the business goals using the standard goals template shown on page 45. The whole team reviewed the business goals with Mayor van Damme and the Office of Management and Budget to verify that we understood the business goals correctly. Our product manager added our business goals to the project wiki so that everyone could read them.

While talking to stakeholders about business goals, you heard many requests for features and a few pain points around quality attributes. Since we have a good handle on the business goals, it will be easier for us to focus our design efforts as we dig for additional architecturally significant requirements.

Next Up

Empathy is the engine that drives design. When we know who our stakeholders are and how they hope the software will help them, we will make better design decisions on their behalf. Business goals are a straightforward way to help the team internalize stakeholders' hopes and dreams for the software.

Knowing who the stakeholders are and understanding the business goals is important but doesn't tell us what the software should do or how it is expected to behave. We capture this information as requirements. Architects need different information than what traditional requirements specifications typically offer. In the next chapter, you'll learn how to look at requirements from the perspective of software architecture.

CHAPTER 5

Dig for Architecturally Significant Requirements

Every design discussion starts with *who*, *what*, and *why*. In Chapter 4, *Empathize with Stakeholders*, on page 39 you learned how to identify *who* is affected by the software system and *why* they care. In this chapter, you'll learn how to define the *what*, the requirements, from the perspective of software architecture.

An *architecturally significant requirement*, or ASR, is any requirement that strongly influences our choice of structures for the architecture. It is the software architect's responsibility to identify requirements with architectural significance. You'll do this by thinking about four categories of requirements:

Constraints Unchangeable design decisions, usually given, sometimes chosen.

Quality Attributes Externally visible properties that characterize how the system operates in a specific context.

Influential Functional Requirements Features and functions that require special attention in the architecture.

Other Influencers Time, knowledge, experience, skills, office politics, your own geeky biases, and all the other stuff that sways your decision making.

Let's take a closer look at these categories of ASRs and learn how to work with stakeholders to define them.

Limit Design Options with Constraints

A *constraint* is an unchangeable design decision you are given or choose to give yourself. Most software systems have only a handful of constraints. All

constraints limit choice, but well-chosen constraints simplify the problem and can make it easier to design a satisficing architecture. Sometimes constraints create a living hell for architects by limiting options so severely we are unable to satisfy other requirements.

Constraints can influence technical or business concerns. Business constraints limit decisions about people, process, costs, and schedule. Technical constraints limit decisions about the technology we may use in the software system. Here are some examples of each:

Technical Constraints	Business Constraints
Programming Language Choice	Team Composition and Makeup
Anything that runs on the JVM.	*Team X will build the XYZ component.*
Operating System or Platform	Schedule or Budget
It must run on Windows, Linux, and BeOS.	*It must be ready in time for the Big Trade Show and cost less than $800,000.*
Use of Components or Technology	Legal Restrictions
We own DB2 so that's your database.	*There is a 5 GB daily limit in our license.*

Capture Constraints as Simple Statements

To capture a constraint, describe the decision and its origin in a brief statement. There are some constraints for the Project Lionheart system in the table on page 51, introduced on page 14.

Constraints, once decided, are 100 percent non-negotiable. Be conservative in accepting constraints. There is a huge difference between *this must be done or you will fail* and *this should be done unless you have a good reason not to do it.*

As the software system emerges, design decisions can become constraint-like. Distinguishing between the constraints we created and the ones we were given becomes more difficult as the system grows. Like barnacles on a ship, software slowly gains cruft and becomes less nimble, less clean, less malleable. Eventually, it may become so difficult to amend the architecture that those early design choices become constraints for future designers.

As constraints emerge, be careful to distinguish the constraints chosen for you from the constraints you give yourself. Though it may be difficult, you always have the option of changing a constraining design decision.

Constraint	Origin	Type	Context
Must be developed as open source software.	Mayor van Damme	Business	The City has an Open Data policy and citizens must have access to source code.
Must build a browser-based web application.	Mayor van Damme	Technical	Decreases concerns about software delivery and maintenance.
Must ship by the end of Q3.	Mayor van Damme	Business	Avoids end of fiscal year budget issues.
Must support latest Firefox web browser.	City IT	Technical	Officially supported browser.
Must be served from a Linux server.	City IT	Technical	City uses Linux and open source where possible.

Define the Quality Attributes

Quality attributes describe externally visible properties of a software system and the expectations for that system's operation. Quality attributes define how well a system should perform some action. These *-ilities* of the system are sometimes called *quality requirements*. Here is a list of some common quality attributes from *Software Architecture in Practice [BCK12]*.

Design Time Properties	Runtime Properties	Conceptual Properties
Modifiability	Availability	Manageability
Maintainability	Reliability	Supportability
Reusability	Performance	Simplicity
Testability	Scalability	Teachability
Buildability or Time-to-Market	Security	

Every architecture decision promotes or inhibits at least one quality attribute. Many design decisions promote one set of quality attributes while inhibiting others that are also important! When this happens, we'll trade one quality attribute for another by choosing a structure for the architecture that favors one quality attribute but harms others.

When digging for ASRs, we'll spend most of our time working with quality attributes. Quality attributes are used throughout the design process to guide technology selection, choose structures, pick patterns, and evaluate the fitness of our design decisions.

Joe asks:
Are Quality Attributes Non-functional Requirements?

Traditional software engineering textbooks usually discuss two classes of requirements. *Functional requirements* describe the behavior of the software system. *Non-functional requirements* describe all system requirements that aren't functional requirements, including what we're calling quality attributes and constraints.

When you are designing a software architecture, it's useful to distinguish between functionality, constraints, and quality attributes because each type of requirement implies a different set of forces are influencing the design. For example, constraints are non-negotiable whereas quality attributes can be nuanced and involve significant trade-offs.

Yes, quality attributes are non-functional requirements, but it is strange to use this term to describe them since quality attribute scenarios (sometimes called quality requirements) have a functional piece to them. Quality attributes make sense only in the context of system operation. In a quality attribute scenario, an artifact's response is the direct result of some function.

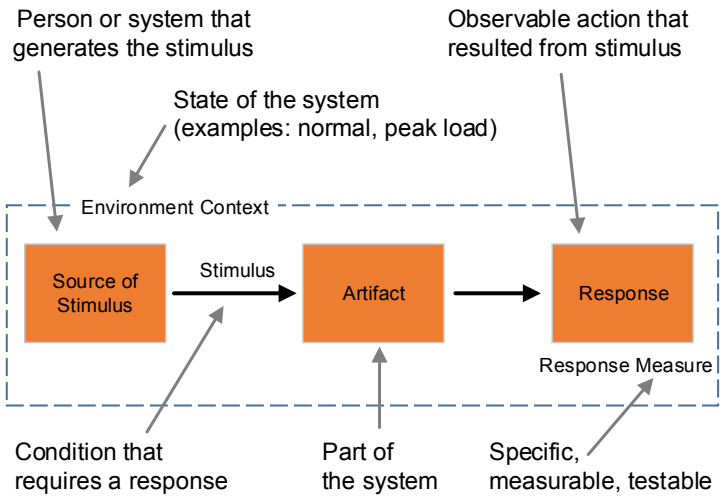
Capture Quality Attributes as Scenarios

A quality attribute is just a word. Scalability, availability, and performance are meaningless by themselves. We need to give these words meaning so we understand what to design. We use a *quality attribute scenario* to provide an unambiguous description of a quality attribute.

Quality attribute scenarios describe how the software system is expected to operate within a certain environmental context. There is a functional component to each scenario—stimulus and response—just like any feature. Quality attributes scenarios differ from functional requirements since they qualify the response using a response measure. It is not enough just to correctly respond. How the system responds is also important. The diagram on page 53 visually depicts the six parts of a quality attribute scenario.

Stimulus The stimulus is an event that requires the system to respond in some way. The stimulus kicks off the scenario and will vary depending on the type quality attribute. For example, the stimulus for an availability might be a node becoming unreachable whereas the stimulus for a modifiability scenario might be a request for a change.

Source The source is the person or system that initiations the stimulus. Examples include users, system components, and external systems.



Artifact The artifact is the part of the system whose behavior is characterized in the scenario. The artifact can be the whole system or a specific component.

Response The response is an externally visible action that takes place in the artifact as a result of the stimulus. Stimulus leads to response.

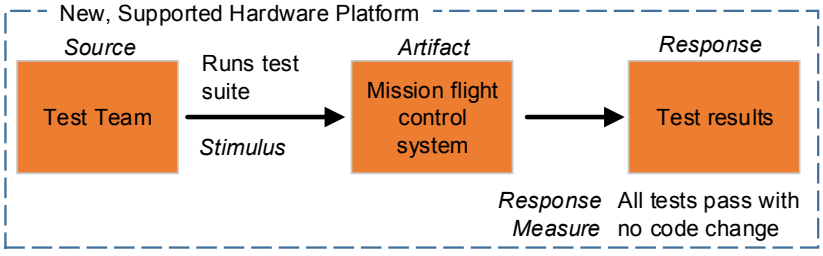
Response Measure The response measure defines the success criteria for the scenario by defining what a successful response looks like. Response measures should be specific and measurable.

Environment Context The environment context describes the operational circumstances surrounding the system during the scenario. The environment context should always be defined even if the context is *normal*. Abnormal contexts, such as peak load or a specific failure condition, are also interesting to consider.

Here is an example portability scenario for an interplanetary robotic explorer based on examples from *the NASA Jet Propulsion Laboratory [WFD16]*.

Portability Scenario for a Mars Rover (via NASA JPL)

Raw scenario: Processors and platforms are typical variation points project to project. Enabling projects to select processors and platforms with minimal effects to applications allows for system optimization.



Notice that the *raw scenario* in our example doesn't mention specific response measures. Raw scenarios are simple descriptions that form the basis for more precise quality attribute scenarios. We call them raw because they need further cooking to become a good scenario. Think of a raw scenario as the start of a conversation.

Specifying all six parts of a formal quality attribute scenario is not always necessary. You can often get by with a simple statement that includes the stimulus, source, response, and response measure. Add the environment whenever the scenario does not describe a *normal* environmental context.

Here are some quality attribute scenarios for the Project Lionheart case study:

Quality Attribute	Scenario	Priority
Availability	When the RFP database does not respond, Lionheart should log the fault and respond with stale data within 3 seconds.	High
Availability	A user's searches for open RFPs and receives a list of RFPs 99% of the time on average over the course of the year.	High
Scalability	New servers can be added during a planned maintenance window (less than 7 hours).	Low
Performance	A user sees search results within 5 seconds when the system is at an average load of 2 searches per second.	High
Reliability	Updates to RFPs should be reflected in the application within 24 hours of the change.	Low
Availability	A user-initiated update (for example, starring an RFP) is reflected in the system within 5 seconds.	Low
Availability	The system can handle a peak load of 100 searches per second with no more than a 10% dip in average response times.	Low
Scalability	Data growth is expected to expand at a rate of 5% annually. The system should be able to grow to handle this with *minimal* effort.	Low

A good-quality attribute scenario communicates the intent of the requirement so anyone can understand it. Great scenarios are precise and measurable. Two people who read the same quality attribute scenario should come away with the same understanding of the system's *scalability* or *performance* or *maintainability*.

Strive for Specific and Measurable Response Measures

To create a response measure, start by estimating potential values based on your own experience. Use a straw man to kick off a conversation with stakeholders (see Activity 9, *Response Measure Straw Man*, on page 219). What if it took nine months to migrate the system to a new microcontroller platform, would that work? How about six months? Eventually, you'll find a response measure that resonates with stakeholders.

Good response measures are testable. Early in the system's life, the architecture might exist only on paper, but it's just a matter of time before you have a running system. If you can't write a test using your scenario, then the scenario does not have a specific, measurable response measure.

Choose Appropriate Response Measures

I was once on a team responsible for building a simulation testbed for a military combat system. The purpose of the testbed was to connect a dozen military bases across the world so we could play simulated war games. To run a test, we would play a scenario that generated fake aircraft. The hardware and software at each site would detect the simulated aircraft, and the combat systems would process the sensor data as if it were from the real world.

The simulation testbed had extremely aggressive latency requirements. If all sites did not receive the same simulation data within a narrow window of time, it would seem as if aircraft were appearing and disappearing from the sky. Even worse, aircraft might be visible only to some sites in the network. Too much latency would invalidate the system tests.

After crunching some numbers, we determined our testbed would need to transfer data faster than the speed of light for everything to work. The performance and availability response measures were nowhere near reality. Once quantum entangled networks become viable, it will be interesting to revisit this problem.

 Get Your Hands Dirty: Refine These Notes into Quality Attribute Scenarios

During a meeting, Project Lionheart stakeholders shared the following statements. For each statement, identify the quality attribute and create a formal, six-part quality attribute scenario.

- There's a small number of users, but when a user submits a question or problem we need to be able to respond quickly, within a business day.
- Releases happen at least once a month. Ideally, we'll ship code as it is ready.

- We need to verify that the RFP index is built correctly. The verification should be automated.
- We need a new, permanent dev team to come up to speed quickly after the current team of contractors we've hired leaves.

Here are some things to think about:

- What quality attribute is suggested by each statement? It's OK to make up an *ility* if it helps describe the concern effectively.
- Are there implied responses or response measures?
- What missing information can you fill in based on your own experiences?

Look for Classes of Functional Requirements

Functional requirements, often captured as use cases or user stories, define the software system's behavior but are only sometimes interesting when designing the architecture. All functional requirements are essential to the success of the software system, but not all system features have architectural significance. When a functional requirement drives architectural decision making, we call it an *influential functional requirement*.

Influential functional requirements can be referred to as *architecture killers*. If your architecture doesn't allow you to implement one of these high-value, high-priority features, you'll be forced to raze your architecture and start over.

Identifying influential functional requirements is equal parts art and science. It becomes easier with experience. Here's how I do it:

1. Start with a notional architecture sketch that summarizes your current thinking about the architecture.
2. Identify general classes of requirements that represent the same type of architectural problem.
3. For each problem class identified, walk through the notional architecture and show how to achieve each requirement group. If it is not immediately obvious how you would implement the feature based on the known coarse-grained requirements, it might have architectural significance.

The goal of step two is to reduce a giant list of functional requirements down to a small number of representative categories. Here are a few strategies:

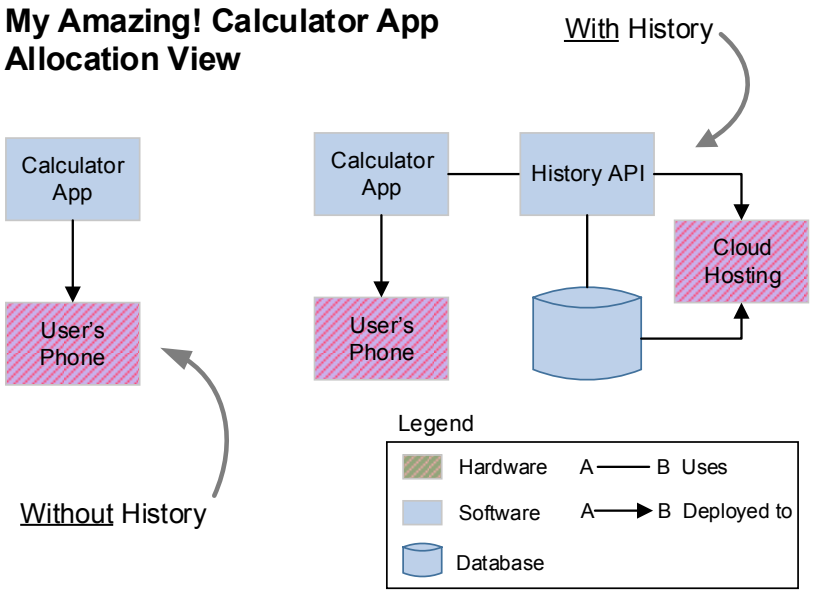
- Look for functional requirements that might be implemented within the same architectural elements. For example, features that require

persistence go in one group whereas features that require user interaction go in another.

- Look for functional requirements that seem difficult to implement. These could be significant to the architecture.
- Look for high-value, high-priority functional requirements.

Here's an example. Recall the simple calculator example on page 10. Adding two numbers together is an important functional requirement but has little influence on the architecture, so let's spice it up a bit. Here's a new feature: *as an Adder User I can review my addition history even if I've lost my phone.* "People love looking at past stuff they've done," the marketing team assures us. "It's going to be A-Mazing!"

Historical information? OK, no problem. We can save the user's actions to a local database. Wait… even if they've lost their phone?!?! Now we need a remote database server, which opens up a ton of new questions. What happens when the user's phone is offline? What about availability? Scalability? Hosting costs? Syncing the app when the schema changes? The list goes on.

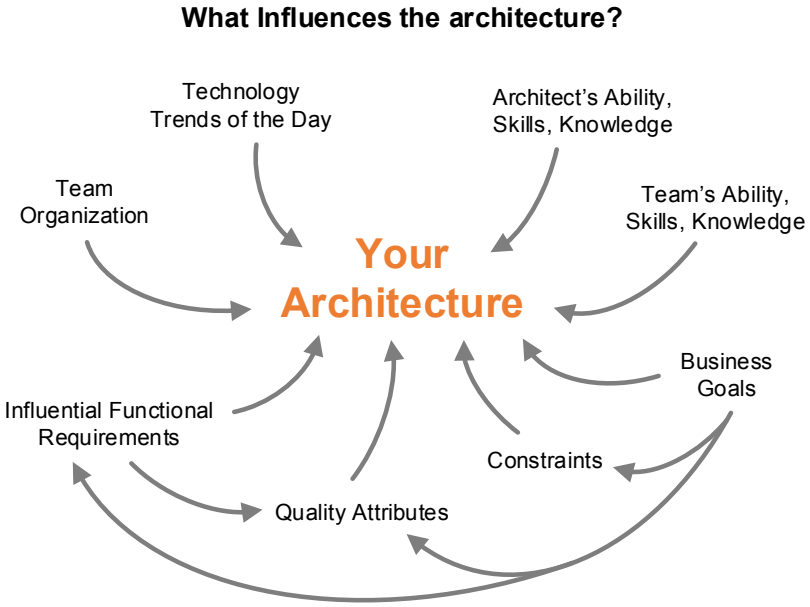


That one seemingly innocent feature request introduces a swirl of complexity. In our simple calculator example, we can reduce any mathematical operation to the same general problem. To solve the newly requested history feature, we need remote storage. This one feature takes the architecture in a new direction that other functional requirements did not.

Reference any influential functional requirements in your architecture documentation but avoid duplicating the requirements engineering effort. Our goal is to call attention to critical features that influence our decision making.

Find Out What Else Influences the Architecture

In addition to ASRs, there is a slew of other factors that will affect the architecture both directly and indirectly. Here is a list of some of the factors that might influence the architecture:



Your skills and experience as an architect determine how you approach design and the architecture options available to you. Your knowledge and your team's knowledge of technology defines your design vocabulary. If all you know is Ruby on Rails, then the chances are good you'll find some way to wedge it into the architecture. When all you have is a hammer, you will find plenty of nails to hit.

Architecture always seems to follow hot technology trends. As new hardware, software, and design paradigms emerge, some will permanently alter the software engineering landscape. Others might just be marketing veneer on old ideas. There's a good chance your architecture is already proudly sporting the design equivalent of a mullet hairstyle.

Learn to Live with Conway's Law

How your team is organized and prefers to collaborate influences the architecture design. Conway's Law, coined by Melvin Conway in 1967 and popularized by

Fred Brooks in the *Mythical Man Month [Bro95]*, describes the relationship between team organization and architecture.

> …organizations which design systems…are constrained to produce designs which are copies of the communication structures of these organizations.

If you have three teams, you'll end up with three components. Communication boundaries among people manifest as element boundaries in the architecture. Conway's Law works both ways. Communication paths designed into the architecture will also influence how you organize your teams. If you want to design the best software possible, then you must be prepared to reorganize your team.

Other influencers are usually only recorded as part of the rationale for design decisions. So many things can influence the architecture that it is practically impossible to document all the potential influencers prior to making design decisions.

Dig for the Information You Need

Architecturally significant requirements are hidden all around us. You'll find ASRs in user stories, implied by a manager's request, and hinted by stakeholders who know what they want but don't quite know how to explain it.

The product backlog contains a treasure trove of ASRs. Quality attributes are implied or assumed in nearly every functional requirement. Sometimes a user story will plainly describe response times, scalability needs, or how to handle failures. Highlight these details as quality attribute scenarios lest they get lost in the feature backlog.

Talk to stakeholders. Find out what worries them. Ask stakeholders what excites them. Share the risks and open questions you see. Here are some additional methods you can use to dig out interesting ASRs:

- Use Activity 3, *Goal-Question-Metric (GQM) Workshop*, on page 199 to connect business goals and quality attribute response measures with concrete data requirements.
- Use Activity 4, *Interview Stakeholders*, on page 202 to uncover quality attribute scenarios and constraints. Interviews work especially well with technical stakeholders.
- Use Activity 5, *List Assumptions*, on page 205 to flush hidden requirements into the open.
- Use Activity 7, *Mini-Quality Attribute Workshop*, on page 210 to quickly and effectively define high-priority quality attribute scenarios. This workshop

works for nearly any kind of project and with stakeholders of different skills and backgrounds.

- Use Activity 24, *Inception Deck*, on page 269 as a checklist for kicking off a new project. Architecture is the main topic for several slides in the inception deck.

Build an ASR Workbook

Once you've identified requirements with architectural significance, record them in an *ASR Workbook*. At the beginning of a new software system, the ASR Workbook is a living document and changes rapidly. As the architecture coalesces, you'll edit the workbook less frequently but reference it more often. Executable tests and source could eventually supplant portions of the ASR Workbook as a source of truth, though the document will remain an important historical record.

The ASR Workbook provides context and information for programmers, testers, and of course, architects. The more people who understand the ASRs, the less architectural oversight will be required.

Here is a sample ASR Workbook outline. Use the outline as a checklist for planning requirements elicitation.

Sample ASR Workbook Outline
Purpose and Scope
Intended Audience
Business Context
Stakeholders
Business Goals
Architecturally Significant Requirements
Technical Constraints
Business Constraints
Quality Attribute Requirements
Top Scenarios
Influential Functional Requirements
Top Users or User Personas
Use Cases or User Stories
Appendix A: Glossary
Appendix B: Quality Attributes Taxonomy

Thijmen says:
Learn to Be an Active Listener

by Thijmen de Gooijer, IT Architect

Understanding your stakeholders and their goals is the first step to successfully delivering value with software. Taking someone else's perspective and showing empathy will help you understand their expectations of your software. Technical training and experience then turn requirements into implementable ideas. However, you will need excellent communication skills to get developers and management to share your vision and turn ideas into code.

One of the most useful communication skills I had to learn is active listening. Hearing what someone says is only the first step. You also have to understand it. Here is a surprisingly challenging exercise that I learned during a course on communication. You can try this exercise with a partner.

Person A tells a story to person B, for example about an achievement or describing a problem they solved. The trick is that person B is not allowed to say a word until person A indicates they are finished. Only then is person B allowed to ask questions to increase understanding of what person A said. The questions cannot be covert feedback or critique. Person B's goal is to reach an understanding. Now reverse roles to experience the other side of the relationship.

Imagine how hard this exercise can be! You probably know a colleague who talks a little bit too much or a shy and quiet intern. How would you ensure that you understand their requirements?

Writing down directly what a stakeholder tells you probably won't lead to an implementable requirement. Human language is messy, complicated, and full of culturally loaded messages—nothing like COBOL, Java, PHP, or Python. Culture does not only differ between countries or religions. Cities, companies, schools, and sports clubs have cultures, which influence how people communicate.

As an active listener, you need empathy to put words into their cultural context and understand them. Remain quiet, don't judge, and ask questions to help you understand.

Communication skills are hard to learn from books, yet I recommend two to assist you on your way to becoming an amazing software architect. *How to Win Friends and Influence People [Car09]* by Dale Carnegie is a classic book that gives actionable guidance on how to build better relationships with people. In *Culture Clash 2: Managing the Global High Performance Team [Zwe13]* Thomas D. Zweifel provides an easy-to-understand framework for identifying and overcoming cultural differences.

Use the ASR Workbook to introduce architectural concepts to your team and stakeholders. Briefly teach readers what business goals, constraints, quality attributes, and influential functional requirements are, and they'll have a finer appreciation for the information in the document.

Project Lionheart: The Story So Far…

During a user experience workshop facilitated by our product manager, you discover dozens of new features. You add these features to the product backlog and make a note in the few functional requirements that seem to have architectural significance. You also jot down several potential constraints to verify with stakeholders.

A few days after the requirements workshop, you facilitate a mini-quality attribute workshop with several stakeholders. During the workshop, you elicit and prioritize nearly two dozen quality attribute scenarios. You don't formally record all the concerns raised during the workshop, but you collaborate with participants to refine the top seven highest-priority scenarios.

Up to this point, our primary focus was to understand the problem. We made several artifacts so we could share what we know about the problem with our stakeholders. You uncover a lot in a few short days on site. Looking at the team's list of open questions, you think we have enough information to embrace the explore mindset and start choosing structures for the architecture.

Next Up

Many different architectures could implement the same set of features. Features alone are not enough information for us to design a software system. It's the architecturally significant requirements, especially quality attributes, that drive architectural decision making.

Solutions flow from our understanding of the problem. We do not need to wait until we understand everything about the problem before thinking about potential solutions. We'd never build anything if we waited to define the whole problem! As you explore solutions, you'll uncover new insights about the problem. Discovering there is more to the problem than you knew is natural and expected. In the next chapter, you'll learn how to use what we currently know about the problem to explore design options and make decisions.