Chapter 5. Identifying Architectural Characteristics

To create an architecture or determine the validity of an existing architecture, architects must analyze two things: architectural characteristics and domain. As you learned in Chapter 4, identifying the correct architectural characteristics ("-ilities") for a given problem or application requires not only understanding the domain problem, but collaborating with stakeholders to determine what is truly important from a domain perspective.

There are at least three places to uncover what architectural characteristics a project needs: the domain concerns, the project requirements, and your implicit domain knowledge. In addition to generic implicit architectural characteristics, such as security and modularity, we've noted that some domains also include implicit architectural characteristics. For example, an architect working on medical software that reads from diagnostics equipment should already understand the importance of data integrity and the potential consequences of losing messages. Architects who work in that domain internalize data integrity, so it becomes implicit in every solution they design.

Extracting Architectural Characteristics from Domain Concerns

Most architectural characteristics come from listening to key domain stakeholders and collaborating with them to determine what is important from a business perspective. While this may seem straightforward, the problem is that architects and domain stakeholders speak different languages. Architects talk about scalability, interoperability, fault tolerance, learnability, and availability; domain stakeholders talk about mergers and acquisitions, user satisfaction, time to market, and competitive advantage. What happens is a "lost in translation" problem where the architect and domain stakeholder don't understand each other. Architects have no idea how to create an architecture to support user satisfaction, and domain stakeholders don't understand why architects focus so much on the application's availability, interoperability, learnability, and fault tolerance.

Fortunately, it's possible to "translate" from domain concerns into the language of architectural characteristics. Table 5-1 shows some of the more common domain concerns and the corresponding "-ilities" that support them. Understanding the key domain goals allows an architect to translate those domain concerns to "-ilities," which then forms the basis for correct and justifiable architecture decisions. For example, is the domain concern of time to market more important than scalability, or should the architect focus on fault tolerance, security, or performance? Perhaps the system requires all these characteristics combined.

Table 5-1. Translating domain concerns into architectural characteristics

Domain concern	Architectural characteristics

Domain concern	Architectural characteristics
Mergers and acquisitions	Interoperability, scalability, adaptability, extensibility
Time to market	Agility, testability, deployability
User satisfaction	Performance, availability, fault tolerance, testability, deployability, agility, security
Competitive advantage	Agility, testability, deployability, scalability, availability, fault tolerance
Time and budget	Simplicity, feasibility

Composite Architectural Characteristics

When translating domain concerns into architectural characteristics, one common pitfall is making false equivalences, such as equating *agility* solely with *time to market*. Agility is an example of a *composite* architectural characteristic: one that has no single objective definition but rather is composed of other measurable things. There's no measure for agility, so architects must ask: what is agility composed of? It includes things like *deployability*, *modularity*, and *testability*, all of which are measurable.

A common antipattern arises when architects focus narrowly on just one part of a composite, often for convenience. This is like forgetting to put the flour in the cake batter. For example, a domain stakeholder might say something like, "Due to regulatory requirements, it is absolutely imperative that we complete end-of-day fund pricing on time."

An ineffective architect might respond by focusing solely on performance, because that seems to be the primary focus of the domain concern. However, that approach will fail, for many reasons:

- It doesn't matter how fast the system is if it isn't available when needed.
- As the domain grows and more funds are created, the system must be able to scale to finish end-of-day processing in time.
- The system must not only be available, it must also be reliable so that it doesn't crash as end-of-day fund prices are being calculated.
- What happens if the system crashes while end-of-day fund pricing is 85% complete? It must be able to recover and restart the pricing where it left off.
- Finally, the system may be fast, but are the fund prices being calculated correctly?

So, in addition to performance, this architect must focus equally on availability, scalability, reliability, recoverability, and auditability.

Many business goals start as composite architectural characteristics. Decomposing them, and giving the resulting characteristics objective definitions, is part of the architects' job. (We'll see the importance of this in Chapter 6.)

Extracting Architectural Characteristics

Most architectural characteristics come from explicit statements in some form of requirements document. For example, lists of domain concerns commonly include explicit numbers of expected users and scale. Other characteristics come from architects' inherent domain knowledge (one of many reasons every architect should know their domain).

For example, suppose you're an architect designing an application that handles class registration for university students. To make the math easy, assume that the school has 1,000 students and 10 hours for registration. Should you make the implicit assumption that the students will distribute themselves evenly over those 10 hours, and design the system to handle a consistent scale? Or, based on your knowledge of university students' habits and proclivities, should you design a system that can handle all 1,000 students attempting to register in the last 10 minutes?

Anyone who understands how much students stereotypically procrastinate knows the answer to this question! Rarely will details like this appear in requirements documents—yet they inform design decisions.

The Origin of Architecture Katas

Over a decade ago, Ted Neward, a well-known architect, devised *architecture katas*, a clever method to allow nascent architects a way to practice deriving architectural characteristics from domain-targeted descriptions. The word *kata* derives from Japanese, where it refers to an individual martial arts training exercise that emphasizes proper form and technique.

> How do we get great designers? Great designers design, of course.
>
> Fred Brooks

Big architectural projects take time, and it's common for architects to design perhaps half a dozen systems in their careers. So how are we supposed to get great architects? The key is giving aspiring architects opportunities to practice their craft. To provide a curriculum, Ted created the first architecture katas site, which your authors Neal and Mark adapted and updated on the companion site for this book at *fundamentalsofsoftwarearchitecture.com*. True to their original purpose, architecture katas provide a useful laboratory for aspiring architects.

Working with Katas

The basic premise is that each kata exercise provides a problem, stated in domain terms, a set of requirements, and some additional context (things that might not appear in the requirements, yet could influence the design). Small teams work on a design (consisting of architectural characteristics analysis and diagrams) for a set time. Then the groups share their results and vote on who came up with the best architecture.

Each kata has predefined sections:

Description

The overall domain problem the system is trying to solve.

Users

The expected number and/or types of users of the system.

Requirements

Domain requirements (as expected from domain users and experts).

Additional context

The authors updated the kata format on the previously mentioned site with an *additional context* section with important considerations, making the exercises more realistic.

We encourage readers to use the site to do their own kata exercise. Anyone can host a brown-bag lunch where a team of aspiring architects can solve a problem. An experienced architect can evaluate the design and trade-off analysis, discussing missed trade-offs and alternative designs either on the spot or after the fact. The designs won't be elaborate because the exercise is timeboxed.

Next, we'll use an architecture kata to illustrate how architects derive architectural characteristics from requirements. Welcome to the *Silicon Sandwiches* kata.

Kata: Silicon Sandwiches

Description

A national sandwich shop wants to enable online ordering in addition to its current call-in service.

Users

Thousands, perhaps one day, millions.

Requirements

- Allow users to place an order and, if the shop offers delivery service, select pickup or delivery.
- Give pickup customers a time to pick up their order and directions to the shop (which must integrate with several external mapping services that include traffic information).
- For delivery service, dispatch the driver with the order to the user.
- Provide mobile device accessibility.
- Offer national daily promotions and specials.
- Offer local daily promotions and specials.
- Accept payment online, at the shop, or upon delivery.

Additional context

- Sandwich shops are franchised, each with a different owner.

- The parent company has near-future plans to expand overseas.
- The corporate goal is to hire inexpensive labor to maximize profit.

Given this scenario, how would you derive architectural characteristics? Each part of the requirement might contribute to one or more aspects of the architecture (and many will not). The architect doesn't design the entire system here—considerable effort must still go into crafting code to solve the domain design (covered in Chapter 8). Instead, look for things that influence or impact the design, particularly structural.

First, separate the candidate architectural characteristics into explicit and implicit characteristics.

Explicit Characteristics

Explicit architectural characteristics appear in a requirements specification as part of the necessary design. For example, a shopping website may aspire to support a particular number of concurrent users, which domain analysts specify in the requirements. Consider each part of the requirements to see if it contributes to an architectural characteristic. But first, consider domain-level predictions about expected metrics, as represented in the Users section of the kata.

One of the first details that should catch your eye is the number of users: currently thousands, perhaps one day, millions (this is a very ambitious sandwich shop!). Thus, *scalability*—the ability to handle a large number of concurrent users without serious performance degradation—is one of the top architectural characteristics. Notice that the problem statement didn't explicitly ask for scalability, but rather expressed that requirement as an expected number of users. Architects must often decode domain language into engineering equivalents.

You'll also probably need *elasticity*—the ability to handle bursts of requests. These two characteristics are often lumped together, but they have different constraints. Scalability looks like the graph shown in Figure 5-1.

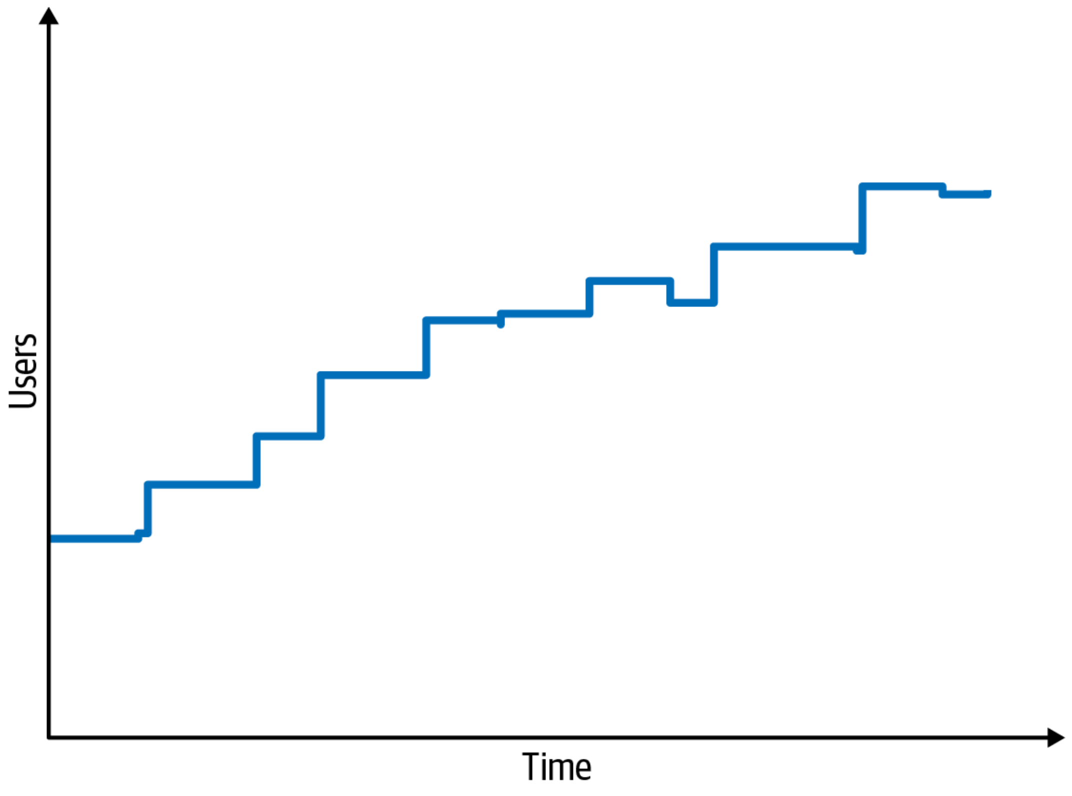


Figure 5-1. Scalability is a measure of the increase of users over time

Elasticity, on the other hand, measures bursts of traffic, as shown in Figure 5-2.

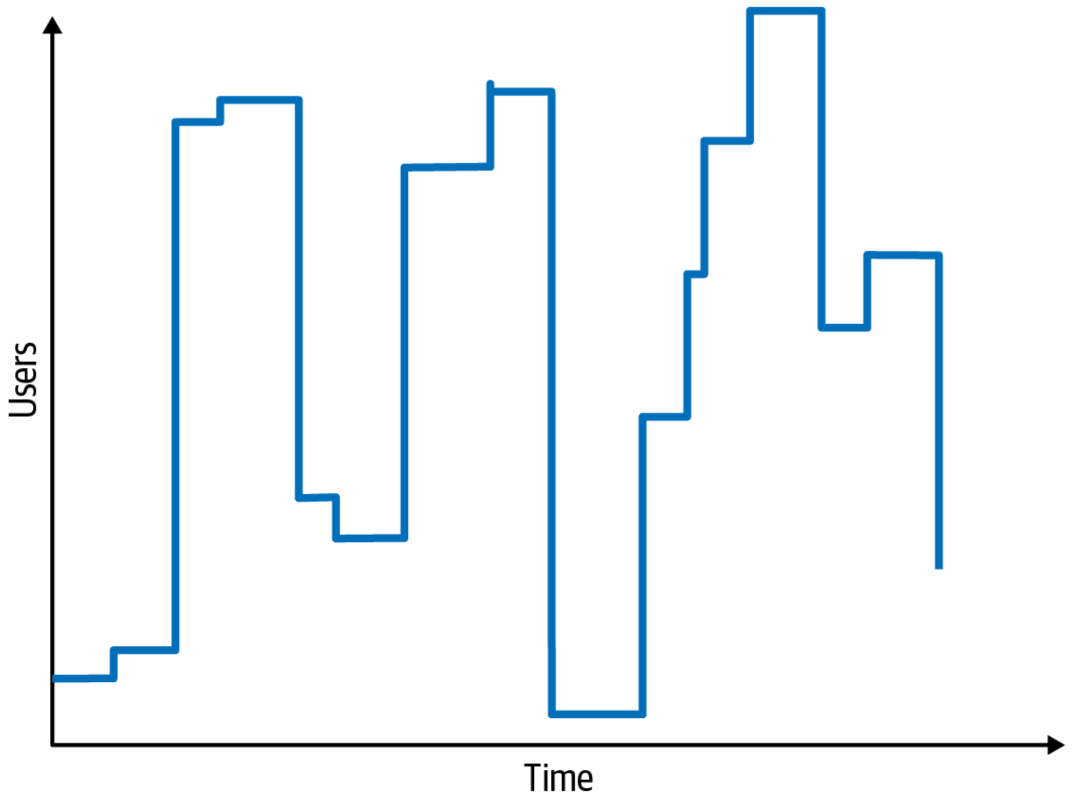


Figure 5-2. Elastic systems must withstand bursts of users

Some systems are scalable but not elastic. For example, a hotel reservation system's number of users, absent special sales or events, is probably predictably seasonal. In contrast, consider a concert-ticket booking system. As new tickets go on sale, fervent fans will flood the site, requiring high degrees of elasticity. Often, elastic systems also need *scalability*: the ability to handle bursts and high numbers of concurrent users.

Elasticity does not appear in the Silicon Sandwiches requirements, but you should still identify it as an important consideration. Requirements sometimes state architectural characteristics outright, but others lurk inside the problem domain. Is a sandwich shop's traffic consistent throughout the day, or does it get bursts of traffic around mealtimes? Almost certainly the latter, so identify this potential architectural characteristic.

Next, consider each of these business requirements in turn to see if it calls for architectural characteristics:

Users place their order, and, if the shop offers delivery service, select pickup or delivery.

> No special architectural characteristics seem necessary to support this requirement.

Pickup customers are given a time to pick up their sandwich and directions to the shop (which must provide the option to integrate with external mapping services that include traffic information).

> External mapping services imply integration points, which may impact aspects such as reliability. For example, say a developer builds a system that relies on a third-party system. If those calls fail, it impacts the reliability of the calling system. Conversely, be wary of overspecifying architectural characteristics. What if the external traffic service is down? Should the Silicon Sandwiches site fail, or should it just offer slightly less efficiency without traffic information? Architects should always guard against building unnecessary brittleness or fragility into designs.

For delivery service, dispatch the driver with the order to the user.

> No special architectural characteristics seem necessary to support this requirement.

Provide mobile device accessibility.

> This requirement will primarily affect the user experience (UX) design of the application, and points toward building either a portable web application or several native web applications. Given the budget constraints and simplicity of the application, it would likely be overkill to build multiple applications, so the design points toward a mobile-optimized web application. Thus, you may want to define some specific performance-related architectural characteristics to optimize page load time and other mobile-sensitive characteristics.
>
> You shouldn't act alone in situations like this. Collaborate with UX designers, domain stakeholders, and other interested parties to vet such decisions. For example, the business may require some behavior that's only possible if you build native applications.

Offer national daily promotions and specials; offer local daily promotions and specials.

> Both of these requirements specify customizability across both promotions and specials. Requirement 1 also implies customized traffic information based on the user's address. Based on both of these requirements, you might

consider customizability as an architectural characteristic. For example, the microkernel architectural style (covered in Chapter 13) supports customized behavior extremely well by defining a plug-in architecture. In this case, the default behavior appears in the core, and developers write the optional location-based custom parts as plug-ins. However, a traditional design can also accommodate this requirement via design patterns (such as the Template Method). This conundrum is common in architecture and requires architects to constantly weigh trade-offs between competing options. We discuss particular trade-offs in more detail in "Design Versus Architecture and Trade-Offs".

Accept payment online, at the shop, or upon delivery.

> Online payments imply a need for security, but nothing in this requirement suggests a particularly high level of security beyond what's implicit. Architects can handle security with either design or architecture, making it a minimal architectural characteristics concern in this application.

Sandwich shops are franchised, each with a different owner.

> This requirement may impose cost restrictions on the architecture. Check the feasibility, applying constraints like cost, time, and staff skill sets, to see if a simple or sacrificial architecture is warranted.

The parent company has near-future plans to expand overseas.

> This requirement implies *internationalization* (often abbreviated as "i18n"). Many design techniques exist to handle this requirement, which shouldn't require special structure to accommodate. This will, however, certainly drive UX decisions.

The corporate goal is to hire inexpensive labor to maximize profit.

> This requirement suggests that usability will be important, but it, too, is more concerned with design than architectural characteristics.

The third architectural characteristic we can derive from the preceding requirements is *performance*: no one wants to buy from a sandwich shop that has poor performance, especially at peak times. However, *performance* is a nuanced concept. What *kind* of performance should you design for? (We cover the various nuances of performance in Chapter 6.)

It's also important to define performance numbers in conjunction with scalability numbers. In other words, establish a baseline of performance without a particular scale, and determine an acceptable level of performance given a certain number of users. Quite often, architectural characteristics interact, forcing architects to define them in relation to one another.

Implicit Characteristics

Many architectural characteristics aren't specified in requirements documents, yet they make up important aspects of the design. One implicit architectural characteristic the system might want to support is *availability*: making sure users can access the site. Closely related to availability is *stability*: making sure the site stays up during interactions. No one wants to purchase from a site that drops connections, forcing them to log in again.

Security appears as an implicit characteristic in every system: no one wants to create insecure software. However, you might give it different priority depending on how critical it is, which illustrates the interlocking nature of our definition. You can consider security an architectural characteristic if it influences some structural aspect of the design and is critical or important to the application.

For Silicon Sandwiches, you might assume that payments should be handled by a third party. Thus, as long as developers follow general security hygiene (not passing credit card numbers as plain text, not storing too much information, and so on), good design in the application will suffice; you shouldn’t need any special structural design to accommodate security. Remember, architectural characteristics are *synergistic*—each architectural characteristic interacts with the others. This is why overspecifying architectural characteristics is such a common pitfall. Overspecifying is just as damaging as underspecifying characteristics because it overcomplicates the system design.

The last major architectural characteristic that Silicon Sandwiches needs to support, *customizability*, is composed of several details from the requirements. Several parts of the problem domain offer custom behavior: recipes, local sales, and directions that may be locally overridden. Thus, the architecture should support custom behavior. Normally, this would fall under application design. However, as our definition specifies, when part of the problem domain relies on custom structure to support it, it moves into the realm of an architectural characteristic. This design element isn’t critical to the application’s success, though. Remember, there are no correct answers in choosing architectural characteristics, only incorrect ones—or:

> There are no wrong answers in architecture, only expensive ones.
>
> One of Mark’s famous quotes

Design Versus Architecture and Trade-Offs

In the Silicon Sandwiches kata, you would likely identify customizability as a part of the system, but the question then becomes: architecture or design? The architecture implies some structural component, whereas design resides within the architecture. For Silicon Sandwiches, you could choose an architecture style like microkernel and build structural support for customization. However, if you choose another style because of competing concerns, developers could implement the customization using the Template Method design pattern, which allows parent classes to define workflows that can be overridden in child classes. Which option is better?

Like everything in architecture, it depends. First, are there good reasons *not* to implement a microkernel architecture (such as performance and coupling)? Second, are other desirable architectural characteristics more difficult in one design versus the other? Third, how much would it cost to support all the architectural characteristics in the architecture versus in the design? This type of trade-off analysis is an important part of an architect’s role.

Above all, it is critical to collaborate with the developers, project manager, operations team, and other co-constructors of the software system. No architecture decision should be made in isolation from the implementation team (which leads to the dreaded Ivory Tower architecture antipattern). In the case of Silicon Sandwiches, the architect, tech lead, developers, and domain analysts should collaborate to decide how best to implement customizability.

You shouldn't stress too much about discovering the exactly correct set of architectural characteristics. Developers can implement functionality in a variety of ways. However, correctly identifying important structural elements may facilitate a simpler or more elegant design. You could design an architecture that doesn't accommodate customizability structurally, instead requiring the design of the application itself to support that behavior (see "Design Versus Architecture and Trade-Offs"). Remember: there is no best design in architecture, only a least worst collection of trade-offs.

Prioritize these architectural characteristics when trying to find the simplest required sets. Once the team has made a first pass at identifying the architectural characteristics, a useful exercise is to try to determine the least important one. If you had to eliminate one, which would it be? Generally, architects are more likely to cull the explicit architectural characteristics, since many of the implicit ones support general success. The way we define what's critical or important to success assists in determining if the application *truly requires* each architectural characteristic. Attempting to determine the least applicable one can help you determine critical necessity.

In the case of Silicon Sandwiches, which of the architectural characteristics we've identified is least important? Again, no absolute correct answer exists. However, in this case, the solution could lose either customizability or performance. We could eliminate customizability as an architectural characteristic and plan to implement that behavior as part of application design. Of the operational architectural characteristics, performance is likely the least critical for success. Of course, that doesn't mean the developers are setting out to build an application that has terrible performance; it simply means this design doesn't prioritize performance over other characteristics, such as scalability or availability.

Limiting and Prioritizing Architectural Characteristics

One tip when collaborating with domain stakeholders to define the driving architectural characteristics: work hard to keep the final list as short as possible. A common antipattern entails trying to design a *generic architecture*: one that supports *all* the architectural characteristics. Each characteristic the architecture supports complicates the overall system design, so supporting too many architectural characteristics leads to greater and greater complexity. And that's before the architect and developers have even started addressing the problem domain—the original motivation for writing the software. Don't obsess over the number of characteristics, but remember the motivation: keeping the design simple.

Case Study: The *Vasa*

The original story of overspecifying architectural characteristics to the point of ultimately killing a project must be the *Vasa*. It was a Swedish warship built between 1626 and 1628 by a king who wanted the most magnificent ship ever created. Up until that time, ships were either troop transports or gunships—the *Vasa* would be both. Most ships had one deck—the *Vasa* had two! All of its cannons were twice the size of those on similar ships. Despite some trepidation, the expert shipbuilders couldn't say no to King Adolphus.

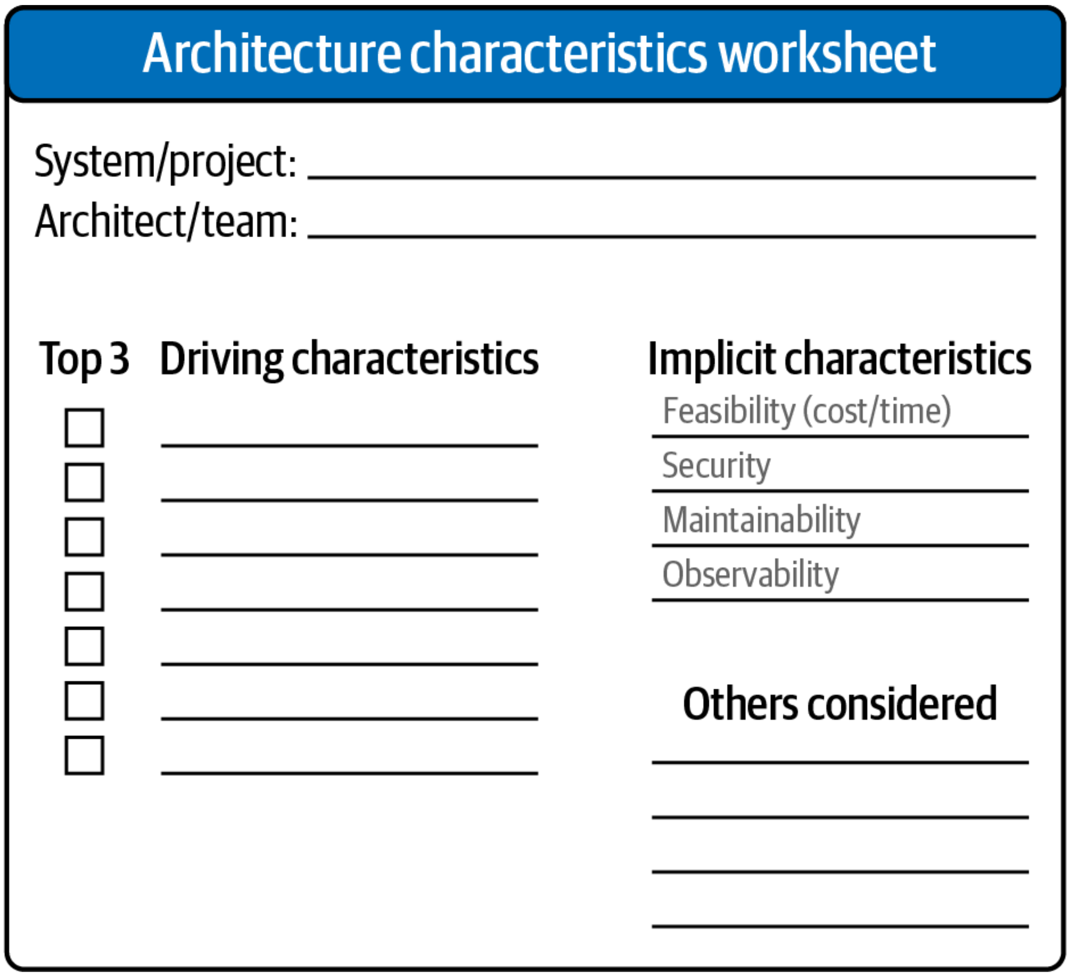
To celebrate finishing construction, the *Vasa* sailed out into Stockholm harbor and shot a cannon salute off one side. Unfortunately, because the ship was top-heavy, it capsized and sank to the bottom. In 1961, salvagers rescued the ship, which now resides in a museum in Stockholm.

Architect and business stakeholder collaboration is crucial during architectural characteristics analysis. However, if the architect provides an extensive list of possible architectural characteristics and asks the business stakeholders which ones they want the architecture to support, what will the answer be every time? *"All of them!"*

Thus, architects need a technique for determining which architectural characteristics drive structural decisions and are critical to success. The authors have developed a number of techniques to facilitate this effort, including an architectural characteristics "worksheet," illustrated in Figure 5-3 (and downloadable at *https://developertoarchitect.com/resources.html*).

The worksheet is meant to be used in an interactive session managed by an architect, who solicits stakeholders' opinions as to the number and details of the required architectural characteristics. On the left, notice the seven slots to list the desired architectural characteristics.

Why seven? Why not? Seriously, an architect needs to restrict the list to some reasonable number—six or eight would also work (although there is some interesting psychology behind the number seven). The second column includes some implicit architectural characteristics; these appear in most systems, but sometimes architects prioritize them as driving concerns of the application that require special design and consideration. In those cases, the architect "pulls" the implicit characteristics into the first column. If the first column fills and a better item appears to displace an existing one, the architect moves it to the "Others Considered" category.



Architecture characteristics worksheet

System/project: ______

Architect/team: ______

Top 3	Driving characteristics
☐	______
☐	______
☐	______
☐	______
☐	______
☐	______
☐	______

Implicit characteristics

- Feasibility (cost/time)
- Security
- Maintainability
- Observability

Others considered

______

______

______

______

Figure 5-3. Architectural characteristics worksheet

The last step is to collaboratively choose the top three highest-priority architectural characteristics, in any order (check the box next to each one). This exercise allows architects to derive a shortened, prioritized list of driving forces they can use to drive design decisions and trade-off analysis.

Many architects and domain stakeholders want to prioritize getting to a final list of architectural characteristics that the application or system must support. While this is certainly a desirable outcome, in most cases, it is a fool’s errand that will not only waste time, but produce a lot of unnecessary frustration and disagreement with the key stakeholders. Rarely will all stakeholders agree on the priority of each and every characteristic. A better approach is to have the domain stakeholders select the top three most important characteristics from the final list (in any order). This makes it much easier to gain consensus and fosters discussions about what is most important. All of this helps the architect analyze trade-offs when making vital architectural decisions.