Chapter 4. Architectural Characteristics Defined

We now delve into the details of structural design, one of the key roles for software architects. This primarily consists of two activities: *architectural characteristics analysis*, covered in this chapter, and *logical component design*, covered in Chapter 8. Architects may perform these two activities in any order (or even in parallel), but they come together at a critical join point.

When a company decides to solve a particular problem using software, it gathers a list of requirements for that system (there are many techniques for eliciting them, as covered in Chapter 8). We'll refer to these requirements as the *problem domain* (or just the *domain*) throughout the book. You learned in Chapter 1 that *architecture characteristics* are the important aspects of a system that are independent from the problem domain and important to the system's success. In this chapter we'll dive deeper into defining that term, as well as specific architectural characteristics.

Architects often collaborate on defining the domain, but must also define, discover, and otherwise analyze all the things the software must do that isn't directly related to the domain functionality: *architectural characteristics*. Architects' role in defining architectural characteristics is part of what distinguishes software architecture from coding and design. They must also consider many other factors in designing a software solution, as illustrated in Figure 4-1.

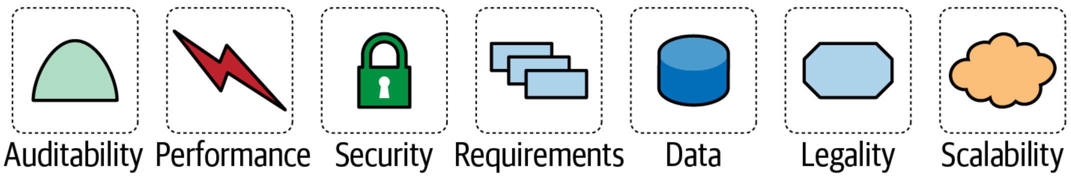

Figure 4-1. A software solution consists of both domain requirements and architectural characteristics

The Longevity of the Term "Non-Functional Requirements"

Many organizations describe architectural characteristics with a variety of terms, including *non-functional requirements*, a term created to distinguish architectural characteristics from *functional requirements*. We dislike that term because it is self-denigrating and has a negative impact from a language standpoint: how do you convince teams to pay enough attention to something that's "non-functional"? Another popular term is *quality attributes*, which we dislike because it implies after-the-fact quality assessment rather than design.

We prefer the term *architectural characteristics* because it describes concerns that are critical to the success of the architecture, and therefore the system as a whole, without discounting the importance of these concerns. In *Head First Software*

Architecture (O'Reilly, 2024), we refer to architectural characteristics as the *capabilities* of the system; in contrast, the domain represents the system's *behavior*.

Sometimes terms get "stuck," and *non-functional requirements* seems to be a particularly sticky one among software architects. It's still common in many organizations. The term first started appearing in software engineering literature in the late 1970s, around the same time as *function point analysis*, an estimation technique that decomposes system requirements into "function points" that each represent some unit of work. Theoretically, teams could add up all the function points at the end of the analysis process and get some insight into the project. Sadly, it provided a veneer of certainty but was plagued by subjectivity, as many estimation schemes are, and is no longer in use.

However, one insight that remains with us from that time is that much of the effort involved to create a system is about that system's *capabilities* rather than its requirements. They called those efforts *non-function points*, which led to the term *non-functional requirements* becoming common.

Architectural Characteristics and System Design

To be considered an architectural characteristic, a requirement must meet three criteria. It must specify a nondomain design consideration, influence some structural aspect of the design, *and* be critical or important to the application's success. These interlocking parts of our definition are illustrated in Figure 4-2, which consists of these three components and a few modifiers.

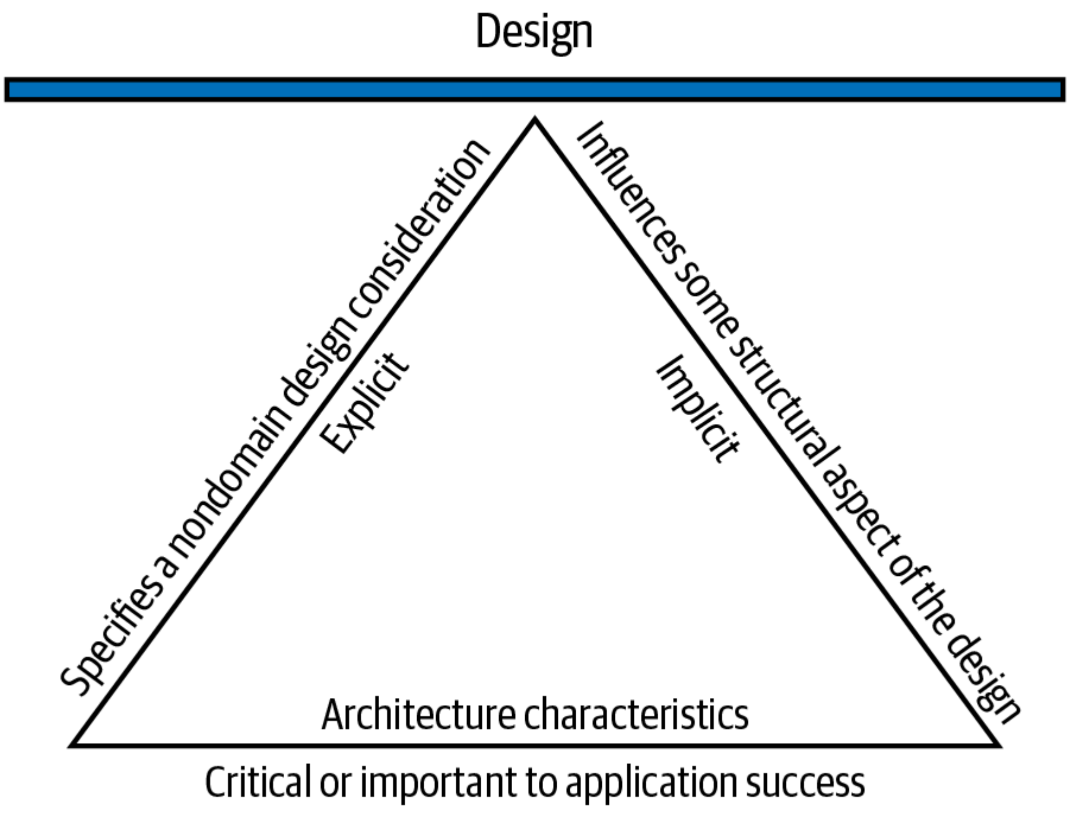
Design
Specifies a nondomain design consideration
Explicit
Influences some structural aspect of the design
Implicit
Architecture characteristics
Critical or important to application success

Figure 4-2. The differentiating features of architectural characteristics

Let's look more closely at these components:

An architecture characteristic specifies a nondomain design consideration.

> Structural design in software architecture consists of two activities by an architect: understanding the problem domain and uncovering what kinds of capabilities the system needs to support to be successful. The domain design considerations cover the behavior of the system, and architectural characteristics define the capabilities. Taken together, these two activities define structural design.
>
> While design requirements specify *what* the application should do, architectural characteristics specify *how* to implement the requirements and *why* certain choices were made: in short, the operational and design criteria for the project to succeed.
>
> For example, specific levels of performance are often an important architectural characteristic, but often don't appear in requirements documents. Even more pertinent: no requirements document actually states that a design must "prevent technical debt," but it is a common design consideration. We cover this distinction between explicit and implicit characteristics in depth in "Extracting Architectural Characteristics from Domain Concerns".

An architecture characteristic influences some structural aspect of the design.

> The primary reason architects try to describe architectural characteristics on projects is to tease out important design considerations. Can the architect implement it via design, or does this architectural characteristic require special *structural* consideration to succeed?
>
> For example, security is a concern in virtually every project, and all systems must take certain baseline precautions during design and coding. However, security rises to the level of an architectural characteristic when the architect determines that the architecture needs special structure to support it.
>
> Consider two common architectural characteristics: security and scalability. Architects can accommodate security in a monolithic system by using good coding hygiene, including well-known techniques such as encryption, hashing, and salting. (Architectural fitness functions, which also fall under this umbrella, are discussed in Chapter 6.) Conversely, in a distributed architecture such as microservices, the architect would build a more hardened service with stricter access protocols—a structural approach. Thus, architects can accommodate security via either design or structure. On the other hand, consider scalability: no amount of clever design will allow a monolithic architecture to scale beyond a certain point. Beyond that, the system must change to a distributed architectural style.
>
> Architects pay close attention to operational architectural characteristics (discussed in "Operational Architectural Characteristics") because they are the characteristics that most often require special structural support.

An architecture characteristic must be critical or important to application success.

> Applications *can* support a huge number of architectural characteristics… but they shouldn't. Each architectural characteristic that a system supports adds complexity to its design. That's why architects should strive to choose the *fewest* possible architectural characteristics rather than the most.

We divide architectural characteristics into implicit versus explicit architectural characteristics. Implicit ones rarely appear in requirements, yet they're necessary for the project to succeed. Availability, reliability, and security underpin virtually all applications, yet they're rarely specified in design documents. Architects must use their knowledge of the problem domain to uncover these architectural characteristics during the analysis phase. For example, a high-frequency trading firm may not have to specify low latency in every system because the architects in that problem domain already know how critical it is. Explicit architectural characteristics appear in requirements documents or other specific instructions.

In Figure 4-2, the choice of a triangle is intentional: each of the definition elements supports the others, which in turn support the overall design of the system. The fulcrum created by the triangle illustrates how these architectural characteristics often interact with one another. This is why architects use the term *trade-off* so much.

Architectural Characteristics (Partially) Listed

Architectural characteristics exist along a broad spectrum of complexity, ranging from low-level code characteristics (such as modularity) to sophisticated operational concerns (such as scalability and elasticity). There is no true universal standard, though people have attempted to codify one. Instead, each organization interprets these terms for itself. Additionally, because the software ecosystem changes so fast, new concepts, terms, measures, and verifications are constantly appearing, providing new opportunities to define architectural characteristics.

While the sheer volume and breadth of architecture characteristics make it hard to quantify them, architects do categorize them. The following sections describe a few such broad categories and provide some examples.

Operational Architectural Characteristics

Operational architectural characteristics cover capabilities such as performance, scalability, elasticity, availability, and reliability. Table 4-1 lists some operational architectural characteristics.

Table 4-1. Common operational architectural characteristics

Term	Definition
Availability	How much of the time the system will need to be available; if that's 24/7, steps need to be in place to allow the system to be up and running quickly in case of any failure.
Continuity	The system's disaster recovery capability.
Performance	How well the system performs; ways to measure this include stress testing, peak analysis, analysis of the frequency of functions used, and response times.
Recoverability	Business continuity requirements: in case of a disaster, how quickly the system must get back online. This includes backup strategies and requirements for duplicate hardware.
Reliability/safety	Whether the system needs to be fail-safe, or if it is mission critical in a way that affects lives. If it fails, will it cost the company large sums of money? This is often a spectrum rather than a binary.
Robustness	The system's ability to handle error and boundary conditions while running, for example, if the internet connection or power

Term	Definition
	fails.
Scalability	The system's ability to perform and operate as the number of users or requests increases.

Operational architectural characteristics overlap heavily with operations and DevOps concerns.

Structural Architectural Characteristics

Architects are responsible for proper code structure. In many cases, the architect has sole or shared responsibility for the code's quality, including its modularity, its readability, how well coupling between components is controlled, readable code, and a host of other internal quality assessments. Table 4-2 lists a few structural architectural characteristics.

Table 4-2. Structural architectural characteristics

Term	Definition
Configurability	How easily end users can change aspects of the software's configuration through interfaces.
Extensibility	How well the architecture accommodates changes that extend its existing functionality.
Installability	How easy it is to install the system on all necessary platforms.
Leverageability/reuse	The extent to which the system's common components can be leveraged across multiple products.
Localization	Support for multiple languages on entry/query screens in data fields.
Maintainability	How easy it is to apply changes and enhance the system.
Portability	The system's ability to run on more than one platform (such as Oracle and SAP DB).
Upgradeability	How easy and quick it is to upgrade to a newer version on servers and clients.

Cloud Characteristics

The software development ecosystem constantly changes and evolves; the most recent excellent example is the arrival of the cloud. When the first edition was published, cloud-based computing existed but wasn't pervasive. Now, most systems have some interaction with cloud-based systems in at least some capacity. A few of these considerations appear in Table 4-3.

Table 4-3. Cloud provider architectural characteristics

Term	Definition
On-demand scalability	The cloud provider's ability to scale up resources dynamically based on demand.
On-demand elasticity	The cloud provider's flexibility as resource demands spike; similar to scalability.
Zone-based availability	The cloud provider's ability to separate resources by computing zones to make for more resilient systems.
Region-based privacy and security	The cloud provider's legal ability to store data from various countries and regions. Many countries have laws governing

Term	Definition
	where their citizens' data may reside (and often restricting it from storage outside their region).

For this book's second edition, we've added a section to each of the architectural style chapters that describes how that style accommodates and facilitates cloud considerations.

Cross-Cutting Architectural Characteristics

While many architectural characteristics fall into easily recognizable categories, others fall outside them or defy categorization, yet form important design constraints and considerations. Table 4-4 describes a few of these.

Table 4-4. Cross-cutting architectural characteristics

Term	Definition
Accessibility	How easily all users can access the system, including those with disabilities like colorblindness or hearing loss.
Archivability	The system's constraints around archiving or deleting data after a specified period of time.
Authentication	Security requirements to ensure users are who they say they are.
Authorization	Security requirements to ensure users can access only certain functions within the application (by use case, subsystem, web page, business rule, field level, etc.).
Legal	The legislative constraints in which the system operates, such as data protection laws like GDPR or financial-records laws like Sarbanes-Oxley in the US, or any regulations regarding the way the application is to be built or deployed. This includes what reservation rights the company requires.
Privacy	The system's ability to encrypt and hide transactions from internal company employees, even DBAs and network architects.
Security	Rules and constraints about encryption in the database or for network communication between internal systems; authentication for remote user access, and other security measures.
Supportability	The level of technical support the application needs; to what extent logging and other facilities are required to debug errors in the system.
Usability/achievability	The level of training required for users to achieve their goals with the application/solution.

Any list of architectural characteristics will necessarily be incomplete; any software project may invent architectural characteristics based on unique factors. Many of the terms we've just listed are imprecise and ambiguous, sometimes because of subtle nuance or a lack of objective definitions. For example, *interoperability* and *compatibility* may appear to be equivalent, and that will be true for some systems. However, they differ because *interoperability* implies ease of integration with other systems, which in turn implies published, documented APIs. *Compatibility*, on the other hand, is more concerned with industry and domain standards. Another example is *learnability*: one definition is "how easy it is for users to learn to use the software," and another definition is "the level at which the system can automatically learn about its environment in order to become self-configuring or self-optimizing using machine learning algorithms.”

Many definitions overlap: *availability* and *reliability*, for instance. Yet consider the internet protocol IP, which underlies TCP. IP is *available* but not *reliable*: packets may arrive out of order, and the receiver may have to ask for missing packets again.

There is no complete list of standards defining these categories. The International Organization for Standards (ISO) publishes a list organized by capabilities that overlaps with our list here, but mainly establishes an incomplete category list. Here are some of the ISO definitions, reworded to update terms and add categories to align with modern concerns:

Performance efficiency

> Measure of the performance relative to the amount of resources used under known conditions. This includes *time behavior* (measure of response, processing times, and/or throughput rates), *resource utilization* (amounts and types of resources used), and *capacity* (degree to which the maximum established limits are exceeded).

Compatibility

> Degree to which a product, system, or component can exchange information with other products, systems, or components and/or perform its required functions while sharing the same hardware or software environment. It includes *coexistence* (can perform its required functions efficiently while sharing a common environment and resources with other products) and *interoperability* (degree to which two or more systems can exchange and utilize information).

Usability

> Users can use the system effectively, efficiently, and satisfactorily for its intended purpose. It includes *appropriateness recognizability* (users can recognize whether the software is appropriate for their needs), *learnability* (how easily users can learn how to use the software), *user error protection* (protection against users making errors), and *accessibility* (make the software available to people with the widest range of characteristics and capabilities).

Reliability

> Degree to which a system functions under specified conditions for a specified period of time. This characteristic includes subcategories such as *maturity* (does the software meet the reliability needs under normal operation), *availability* (software is operational and accessible), *fault tolerance* (does the software operate as intended despite hardware or software faults), and *recoverability* (can the software recover from failure by recovering any affected data and reestablish the desired state of the system).

Security

> Degree to which the software protects information and data so that people or other products or systems have the degree of data access appropriate to their types and levels of authorization. This family of characteristics includes *confidentiality* (data is accessible only to those authorized to have

access), *integrity* (the software prevents unauthorized access to or modification of software or data), *nonrepudiation* (can actions or events be proven to have taken place), *accountability* (can user actions of a user be traced), and *authenticity* (proving the identity of a user).

Maintainability

Represents the degree of effectiveness and efficiency to which developers can modify the software to improve it, correct it, or adapt it to changes in environment and/or requirements. This characteristic includes *modularity* (degree to which the software is composed of discrete components), *reusability* (degree to which developers can use an asset in more than one system or in building other assets), *analyzability* (how easily developers can gather concrete metrics about the software), *modifiability* (degree to which developers can modify the software without introducing defects or degrading existing product quality), and *testability* (how easily developers and others can test the software).

Portability

Degree to which developers can transfer a system, product, or component from one hardware, software, or other operational or usage environment to another. This characteristic includes the subcharacteristics of *adaptability* (can developers effectively and efficiently adapt the software for different or evolving hardware, software, or other operational or usage environments), *installability* (can the software be installed and/or uninstalled in a specified environment), and *replaceability* (how easily developers can replace the functionality with other software).

The last item in the ISO list addresses the functional aspects of software:

Functional suitability

This characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. This characteristic is composed of the following subcharacteristics:

Functional completeness

Degree to which the set of functions covers all the specified tasks and user objectives.

Functional correctness

Degree to which a product or system provides the correct results with the needed degree of precision.

Functional appropriateness

Degree to which the functions facilitate the accomplishment of specified tasks and objectives.

However, we do not believe that functional suitability belongs in this list. It does not describe architectural characteristics but rather the motivational requirements to build the software. This illustrates how thinking about the relationship between architectural characteristics and the problem domain has evolved. We cover this evolution in Chapter 7.

The Many Ambiguities in Software Architecture

A source of consistent frustration among architects is the lack of clear definitions for so many critical things, including the activity of software architecture itself! The absence of a standard leads companies to define their own terms for common things. This often leads to industry-wide confusion, because architects either use opaque terms or, worse yet, the same terms for wildly different meanings.

As much as we'd like, we can't impose a standard nomenclature on the software development world. However, to help avoid terminology-based misunderstandings, domain-driven design advises organizations to establish and use a *ubiquitous language* among employees. We follow and recommend that advice.

Trade-Offs and Least Worst Architecture

We said earlier that architects should support only those architectural characteristics that are critical or important to the system's success. Systems can only support a few of the architectural characteristics we've listed, for a variety of reasons. First, support for an architectural characteristic is rarely free. Each supported characteristic requires design effort from the architect, effort from developers to implement and maintain it, and perhaps structural support.

Second, architectural characteristics are synergistic with each other *and* the problem domain. As much as we would prefer otherwise, each design element interacts with all the others. For example, taking steps to improve *security* will almost certainly negatively impact *performance*: the application must do more on-the-fly encryption, indirection (for hiding secrets), and other activities that can degrade performance. Airplane pilots often struggle when learning to fly helicopters, which have controls for each hand and each foot. Changing one control impacts all of the others because they are synergistic. Flying a helicopter is a balancing exercise, which nicely describes the trade-off process when choosing architectural characteristics; they are similarly synergistic with both other architectural characteristics and the domain design: changing one often entails changing another. Like our belabored helicopter pilot, architects must learn to juggle interlocking items.

Third, as we discussed previously, the lack of standard definitions for architectural characteristics means that organizations struggle with ambiguity. While the industry will never be able to create an immutable list of architectural characteristics (because new ones constantly appear), each organization can create its own list (or *ubiquitous language*) with objective definitions.

Finally, not only is the number of architectural characteristics constantly increasing, but the number of *categories* has increased over the last decade as well. For example, a few decades ago, architects cared little for operational concerns, which were considered a separate "black box." However, with the increased popularity of architectures like microservices, architects and operations must collaborate more intensely and frequently. As software architecture becomes more complex, it tends to entangle with other parts of the organization.

Thus, it’s very rare for architects to be able to design a system and maximize every single architectural characteristic. More often, the decisions come down to trade-offs between several competing concerns.

Tip

Never strive for the *best* architecture; aim for the *least worst* architecture.

Trying to support too many architectural characteristics leads to generic solutions that attempt to solve every business problem. The design quickly becomes unwieldy, so such architectures rarely work.

Strive to design architecture that is as iterative as possible. The easier it is to change the architecture, the less everyone needs to stress about discovering the exact correct thing in the first attempt. One of the most important lessons of Agile software development is the value of iteration; this holds true at all levels of software development, including architecture.