5
Designing Software Architectures

Software architecture design is a key step in building successful software systems, and this chapter begins by exploring what software architecture design is, and why it is important in a software project.

There are two main approaches to architecture design: the top-down and bottom-up approaches. Each approach has advantages and disadvantages, and in this chapter you will learn how to select the best approach for a given project. Designing a software and architecture can be challenging, but we will take a look at design principles and existing solutions that can be leveraged in a design.

Architecture design processes provide guidance to software architects to ensure that a design satisfies requirements, quality attribute scenarios, and constraints. This chapter covers the activities that are typically performed as part of an architecture design process and then provides an overview of four processes: **attribute-driven design** (**ADD**), **Microsoft's technique for architecture and design**, the **architecture-centric design method** (**ACDM**), and the **architecture development method** (**ADM**).

The chapter will conclude by explaining how to use an architecture backlog to prioritize work and track the progress of architecture designs.

In this chapter, we will cover the following topics:

- Software architecture design
- The importance of software architecture design
- Top-down versus bottom-up design approaches
- Greenfield versus brownfield software systems
- Architectural drivers
- Leveraging design principles and existing solutions
- Documenting the software architecture design

- Using a systematic approach to software architecture design
- Attribute-driven design (ADD)
- Microsoft's technique for architecture and design
- Architecture-centric design method (ACDM)
- Architecture development method (ADM)
- Tracking the progress of the software architecture's design

Software architecture design

Software architecture design involves making decisions in order to satisfy functional requirements, quality attributes, and constraints. It is a problem-solving process that leads to the creation of an architecture design.

Software architecture design comprises defining the structures that will make up the solution and documenting them. The structures of a software system are made up of elements, and the relationships between the elements. The properties and behaviors of the elements that are publicly exposed, through an interface, should be identified as part of the design. The design allows you to understand how the elements behave and interact with each other. Private implementations of the elements are not architecturally significant and need not be considered as part of the design.

The software architecture design serves as technical guidance for development and typically occurs iteratively until the initial architecture is at a point where the development team can begin their work. Once an initial architecture is designed, it can continue to evolve as development is taking place. For example, additional design iterations may occur to refactor an architecture to fulfill new requirements or quality attributes.

Software architecture design is a creative process. Software architects have the privilege of coming up with solutions to complex problems and can use creativity to do it. It can be one of the most fun and rewarding parts of a software project.

Making design decisions

The set of software requirements consists of a series of design issues that must be solved. For each of these design issues, such as providing certain business functionality, respecting a particular constraint, meeting performance objectives, or providing a certain level of availability, there may be numerous ways to solve the problem. You will need to consider the strengths and weaknesses of these alternatives in order to select the most appropriate choice.

A large part of software architecture design is making design decisions to resolve issues so that a solution can be implemented. As the software architect, you will be leading the decision-making process.

It is a collaborative process, and usually the best designs incorporate knowledge and feedback from multiple people, such as other software architects and experienced developers. Joint designs and reviewing the architecture with others is beneficial in coming up with a solid software architecture design.

The result of the design is a set of decisions that shape your software architecture. The design is documented in artifacts that can be used for the implementation of a solution.

Software architects should keep in mind that a decision that is made for one design issue may affect another one. This is why software architecture design is an iterative process. Each decision for a design issue may not be optimal for another issue, but the overall solution must be acceptable by satisfying all of the requirements.

Perfect is the enemy of good, an aphorism that has its origins in the thoughts of the French philosopher, Voltaire, and others, is applicable to software architecture design. A completed design may not be perfect, as there will be conflicting requirements that need to be met, and trade-offs made in order to meet them. If the design satisfies all of the requirements, then it is a good one, even if it is not perfect.

Software architecture design terms

Before we go any further, let's define some of the terms that we will be using while detailing the process of software architecture design. These terms can vary, depending on the organization and the team. Regardless of the terms used, the important thing is that they are used consistently by the team members and that they are understood by all team members.

For the purposes of this book, we'll be using the terms structure, element, system, subsystem, module, and component.

Structure

Structures are groupings of, and relations between, elements. Anything that is complex and made up of elements can be referred to as a structure. We previously defined software architecture, in part, by saying it is made up of the structures, their elements, and the relationships of those elements with each other.

Element

An element is a generic term that can be used to represent any of the following terms: system, subsystem, module, or component. If we want to refer to pieces of a software application in a general way, we can refer to them as elements.

System

The software system represents the entire software project, including all of its subsystems. A system consists of one or more subsystems. It is the highest level of abstraction in a software architecture design.

Subsystem

Subsystems are logical groupings of elements that make up a larger system. The subsystems can be created in a variety of ways, including partitioning a system by functionality.

Although they do not have to be, subsystems can represent standalone software applications. An overall software system may be composed of multiple subsystems, and any number of them might be a standalone application. These standalone applications can be external applications that were not developed by the organization.

Organizing a larger software system into subsystems lowers complexity, and allows for software development to be better managed. In some cases, one or more development teams may be formed for each subsystem. Each subsystem is made up of one or more modules.

Module

Modules, like subsystems, are logical groupings of elements. Each module is contained within a subsystem and consists of other modules and/or components. They are typically focused on a single logical area of responsibility.

Development teams assigned to a particular subsystem will be responsible for the modules that make up that subsystem.

Component

Components are execution units that represent some well-defined functionality. They typically encapsulate their implementation and expose their properties and behaviors through an interface.

Components are the smallest level of abstraction and typically have a relatively small scope. Components can be grouped together to form more complex elements, such as modules.

The importance of software architecture design

A software architecture is the foundation of a software system. The design of the architecture is significant to the quality and long-term success of the software. A proper design determines whether the requirements and quality attributes can be satisfied.

There are a number of reasons why a good software architecture design is critical to building useful software. In this section, we will explore the following reasons:

- Software architecture design is when key decisions are made regarding the architecture.
- Avoiding design decisions can incur technical debt.
- A software architecture design communicates the architecture to others.
- The design provides guidance to the developers.
- The impact of the software architecture design is not limited to technical concerns. It also influences the non-technical parts of the project.

Making key decisions

It is during software architecture design that key decisions are made that will determine whether requirements, including quality attributes, can be satisfied. Software architecture enables or inhibits quality attributes, so design decisions play a large role in whether or not they can be met.

Some of the earliest decisions are made during design. If these decisions need to change, it is easier and less costly to change architectural decisions early, before coding has even begun, than to make changes later.

Avoiding design decisions can incur technical debt

Critical decisions are made during the design, and for that reason, there is a cost to either delaying a design decision or not making one at all. Delaying or avoiding certain design decisions can incur technical debt.

Technical debt is similar to financial debt. In the context of design, it is the cost and effort for the additional work that will be necessary later due to decisions that are made now, or because decisions have not been made.

In addition to delaying or avoiding decisions, a decision may be made knowing that it will cost some amount of technical debt. As a software architect, you may decide to take an easier route to a solution, incurring technical debt, even though there is a better solution. As is the case with financial debt, technical debt is not always a bad thing. Sometimes, you will want to pay a debt later in order to get something now. For example, designing a better long-term solution may take more time and effort, and you may decide on a solution that takes less time in order to get the software in production to take advantage of a market opportunity.

It can be difficult to measure the impact of technical debt accurately. Keep in mind that in addition to the time and effort that might be required later to make up for a decision that is made or avoided now, technical debt can have other negative repercussions. For example, a design that is not optimal, leading to lower levels of modifiability and extensibility, can hinder the team's ability to deliver other functionality. This is an additional cost that should be added to the technical debt.

The software architect needs to take all of these factors into consideration when deciding whether or not to incur a technical debt.

Communicating the architecture to others

The results of the architecture design allows you to communicate the software architecture to others. There will be a variety of people who will potentially be interested in the design of the architecture.

The design will also improve cost and effort estimates since it influences what tasks will be required for implementation. Understanding the nature of the work that lies ahead and what types of tasks will be needed to complete the project will assist project managers with their planning. Being able to estimate cost, effort, and the quality attributes that will be met is also useful for project proposals.

Providing guidance to developers

A software architecture design provides guidance to the development team, by steering implementation choices as well as providing training on the technical details of the project.

The design imposes implementation constraints, making it important for coding tasks. Knowing the software architecture design helps developers be aware of the implementation choices available to them, and minimizes the possibility of making an incorrect implementation decision.

It can also be used as training for developers. At the start of the project, the development team will need to understand the design decisions that have been made, and the structures that have been designed. Creating detailed designs for components and implementing them requires an understanding of the architecture design. If new developers join the team later, they can also use the architecture design as part of their onboarding.

Influencing non-technical parts of the project

Another reason that software architecture design is important is the fact that design decisions affect aspects of the software project other than the architecture. For example, certain architecture design decisions could affect the purchasing of tools and licenses, the hiring of team members, the organization of the development environment, and how the software will eventually be deployed.

Top-down versus bottom-up design approaches

There are two fundamental approaches to the design of software architecture. One is a top-down design approach, and the other is a bottom-up approach. These strategies apply to a variety of disciplines, including software architecture design. Let's look at both of them in more detail.

Top-down approach

A **top-down approach** starts with the entire system at the highest level, and then a process of decomposition begins to work downward toward more detail. The starting point is the highest level of abstraction. As decomposition progresses, the design becomes more detailed, until the component level is reached.

While the detailed design and implementation details of the components are not part of the architecture design, the public interfaces of the components are part of the design. It is the public interfaces that allow us to reason about how components will interact with each other.

A design using the top-down approach is typically performed iteratively, with increasing levels of decomposition. It is particularly effective if the domain is well understood.

This systematic approach has been favored by enterprises since it can handle large and complex projects and because the method of design is planned. A systematic approach to architecture design is attractive to enterprises because it can help with time and budget estimates. However, a strict top-down approach, which requires a lot of upfront architecture design, has become less common in modern software architecture.

Advantages of the top-down approach

There are a number of benefits to using a top-down approach. It is a systematic approach to design and breaks the system down into smaller parts. As a system is decomposed, it lends itself well to the division of work. On larger projects with multiple teams, this work can be divided among the teams.

As further decomposition takes place, tasks can be created for individual team members. This supports project management in the assignment of tasks, scheduling, and budgeting. This type of ability to plan is attractive to enterprises. The management teams of organizations may prefer, or even insist on, a top-down approach. Earlier in the book, we discussed how, as a software architect, you may be asked to assist with project estimates, and a top-down approach will allow you to do that with more accuracy.

Although this approach works well on both small and large projects, it can be particularly useful for large projects. By decomposing a system into smaller components, a large project can become more manageable as the size and complexity of each component is reduced.

Disadvantages of the top-down approach

A strictly top-down approach runs the risk of a **big design up front** (**BDUF**), sometimes referred to as a **big up-front design** (**BUFD**). Software is complex and it can be difficult to create the entire architecture up front. Design flaws or missing functionality in the architecture may not be uncovered until later in the process, when components are designed or implemented. If architecture changes are required in higher levels of the architecture after some work is already completed, it will be more difficult to make the modifications.

A top-down approach works best when the domain is well understood, which is not always the case. Plenty of projects begin without the domain being fully understood. Even when it is understood, stakeholders and users can sometimes be unclear as to what the software should do, and how it should work.

If multiple teams are working on a project, each responsible for a particular subsystem or module, knowledge sharing and reuse can be difficult with this approach. Each team can work independently of the others, which has its advantages, but it does not facilitate the sharing of code or knowledge. The software architect may have to recognize areas of reuse, abstract them out, and communicate them to the teams. Another way to mitigate this issue is to provide opportunities and collaboration tools for teams to communicate with each other.

If you use the top-down approach, be careful not to become an ivory tower architect. If you design the higher levels of an architecture and then hand them off to developers to handle the lower-level detailed design, it is easy to become disengaged. As much as your organization and the project permits, make an effort to stay involved with the team. If architectural changes are required later, you will already be familiar with the ongoing implementation, which will help you to make the correct changes.

Bottom-up approach

In contrast with the top-down approach, the **bottom-up approach** begins with the components that are needed for the solution, and then the design works upward into higher levels of abstraction. Various components can then be used together, like building blocks, to create other components and eventually larger structures. The process continues until all the requirements have been met.

Unlike the top-down approach, which begins with the high-level structure, there is no up-front architecture design with the bottom-up approach. The architecture *emerges* as more work is completed. Hence, this is sometimes referred to as *emergent design* or *emergent architecture*.

The bottom-up approach does not require that the domain be well-understood, as the team only focuses on a small piece at a time. The system grows incrementally as the team learns more about the problem domain as well as the solution.

Advantages of the bottom-up approach

One advantage of a bottom-up approach is the greater level of simplicity. The team only has to focus on individual pieces and builds only what it needs for a particular iteration.

This approach works well with agile development methodologies. With an iterative approach that handles change, refactoring can take place to add new functionality or to change existing functionality. Each iteration ends with a working version of the software until eventually the entire system is built. Agile practices, such as automated unit testing and continuous integration, are encouraged and can lead to higher quality software.

A bottom-up approach avoids the possibility of a big design up front, which can lead to overdesigning a solution. Some in the agile community feel that a lot of design effort up front is wasted time and that an emergent design, or **no design up front** (**NDUF**), would be more effective.

The bottom-up approach allows the development team to begin coding very early in the process, which also means testing can occur earlier. This includes automated unit testing as well as manual testing by team members such as QA analysts and other users. Getting feedback earlier in the process allows the team to identify any necessary changes earlier.

This approach facilitates code reuse. As the team is focused on a limited number of components at any given time, recognizing opportunities for reuse becomes easier.

Disadvantages of the bottom-up approach

A bottom-up, or emergent, approach, assumes that change is cheap. Agile methodologies and practices provide an approach that anticipates change and can adapt to it. However, depending on the nature of the change, refactoring software architecture design can be very costly.

A bottom-up approach, with no initial architecture, can lead to lower levels of maintainability. With the refactoring that may be necessary with this approach, issues can arise. If the team is not diligent, this problem can become worse over time.

The entire scope of work may not be known when using this approach. This makes it more difficult to plan and estimate the entire project, which may be unacceptable for enterprise software.

One of the disadvantages of the top-down approach is that design flaws may not be detected until later, leading to costly refactoring. However, just because there is no initial design with the bottom-up approach does not make it immune to uncovering design flaws later in the project. It may not be until after the architecture *emerges* that certain design flaws become apparent.

Which approach should I use?

There are certain factors to consider when deciding whether the top-down or bottom-up approach is better for a software project. Software architects may find it advantageous to use a top-down approach if more than one of the following is true:

- The project is large in size
- The project is complex
- Enterprise software is being designed for an organization
- The team is large, or there are multiple teams that will be working on the project
- The domain is well-understood

It may be more appropriate to use a bottom-up approach if more than one of the following is true:

- The project is small in size
- The project is not very complex
- Enterprise software for an organization is not being designed
- The team is small, or there is only a single team
- The domain is not well-understood

Taking an extreme approach, such as doing a big upfront architecture design or no architecture design at all, is typically not ideal. Although some situations will lead you to select a top-down or bottom-up approach, software architects should also consider using a combination of the two approaches. In this way, you may be able to realize some of the benefits of both approaches, while minimizing the drawbacks.

In the beginning of the project, rather than starting to code immediately, a top-down approach will provide the opportunity to spend at least some time thinking about the overall structure of the design. The design of a high-level architecture provides some structure that can then be leveraged for further design and development.

A high-level architecture design can be used to define and organize the teams, and provides details to project management so that they can perform resource allocation, scheduling, and budget planning. As a software architect, you may be asked to provide input regarding these types of project management activities, and having at least a high-level architecture will assist you in such tasks.

Those who advocate for a strictly bottom-up approach, in which the architecture emerges from implementation, tend to think that software architecture inhibits agility and the ability to make changes to the software. However, as was mentioned in `Chapter 1`, *The Meaning of Software Architecture*, a good software architecture actually facilitates making changes as well as managing them. A good architecture allows you to understand what it would take to make a particular change.

Using a top-down approach for part of the design does not require a big upfront design. You can focus on architecturally significant design issues, and once a high-level architecture is established, you can employ a bottom-up approach. Components and modules based on the high-level architecture can then be designed and implemented.

The quality of the architecture design is not solely dependent on selecting the correct approach. The correct design decisions must be made during the design process, as both a top-down as well as a bottom-up approach can lead to poor architecture designs. A design created with a top-down approach can miss key requirements, which may lead to costly architectural refactoring. A design created with a bottom-up approach may require substantial refactoring while the team figures out how the software system should be structured.

No single approach is applicable to all situations. Each project, organization, and team is different, so the decision of which approach to take will vary. Even with a hybrid approach, the amount of upfront architecture design that is necessary will vary, so it is about determining how much design is needed. That is part of the challenge of being a software architect. Good architects eventually learn how much design is appropriate for a given situation.

Greenfield versus brownfield software systems

When you are starting the design process, one of the first considerations is whether you are designing a greenfield or a brownfield system. The terms **greenfield** and **brownfield** are used in a number of disciplines. It is an analogy to a construction project, and whether it will begin on *greenfield* land, as in land that is undeveloped, or *brownfield* land, referring to land that was previously developed but is not currently in use.

Greenfield systems

A **greenfield software system** is a completely new software application, one in which you can start with a clean slate. There are no constraints based on any prior work. A greenfield system can be designed for a well-understood domain or for a novel domain.

A well-understood domain is one that is mature, and the possibilities for innovation are very limited. Examples include Windows desktop applications, standard mobile applications, and enterprise web applications. There will be existing frameworks, tools, and sample architectures for the software that you need to build. The software architectures of existing applications can be used as a guide.

It will be more likely that you are developing software for a well-understood domain, and the benefit is that there will be a tremendous amount of knowledge that you can leverage from the experience of those who have built similar applications.

A greenfield system for a novel domain is also a new software application that does not need to take into consideration any prior work. The difference between a greenfield system for a mature domain and one for a new domain lies in the fact that a new domain is not as well understood, and requires a lot more innovation.

Unlike a well-understood domain, you will not find as much supporting information for a new domain. Rather than relying on a plethora of reference architectures or referring to a large knowledge base, you will find yourself spending time building prototypes to test out your solutions.

For novel domains, it may be beneficial to design a *throwaway prototype* initially. These are prototypes of some piece of a software system so that you can test it out, such as getting feedback from users or testing quality attributes. They will help you to gain an understanding of what will make a viable solution for a novel domain.

Throwaway prototypes are not built for long-term use, hence the term throwaway, so qualities such as maintainability and reusability are not the focus of such prototypes. If you are using new technologies, or technologies that are not already familiar to you, a prototype can be a good way to try out a solution.

Brownfield systems

A **brownfield software system** is an existing software system. If changes to an existing system require architectural changes, architecture design will be needed. Modifications may be necessary for purposes such as correcting defects, implementing new functionality, or changing existing functionality.

Architectural changes may also be performed on existing software to improve it in some way without changing any functionality. For example, an architecture for an existing software system might be refactored to improve a particular quality attribute. Most of the time, work on brownfield systems does not involve wholesale changes in the overall architecture unless major rework is required.

One of the crucial first steps for the software architecture design of brownfield systems is to gain an understanding of the existing architecture. You need to understand the overall structure, the elements, and the relationships between those elements. From there, the design is not so different from a greenfield system that has been through some iterations to establish an initial architecture.

We will explore the topic of architecture for legacy systems in `Chapter 14`, *Architecting Legacy Applications*.

Architectural drivers

Architectural drivers are considerations that need to be made for the software system that are architecturally significant. They *drive* and guide the design of the software architecture. Architectural drivers describe what you are doing and why you are doing it. Software architecture design satisfies architectural drivers.

Architectural drivers are inputs into the design process, and include:

- Design objectives
- Primary functional requirements

- Quality attribute scenarios
- Constraints
- Architectural concerns

Design objectives

Design objectives focus on the purpose of the specific architecture design. For the particular design in question, what are the reasons behind why the software is being designed?

The design objectives influence the design and are therefore one of the architectural drivers. A common design objective is to design an architecture for a solution, prior to development. The overall objective is to facilitate the implementation of a solution that will satisfy requirements.

This type of design objective might be for a greenfield or a brownfield type of system. As we already explored, the differences between these types of systems might lead you to focus on different design objectives.

Designing a software architecture for development is not the only type of design objective. As a software architect, you may find yourself involved with project proposals. For such pre-sales activity, the design objective may focus on coming up with the software's capabilities, the possible timeframe for delivery, a breakdown of work tasks, and the feasibility of the proposed project. If this is the purpose of the design, then this type of initial design will not be nearly as detailed as one that you are designing for development.

For the purposes of a project proposal, it will not be necessary to be as detailed as you would be in preparation for development. You may be required to produce a design for a project proposal in a short amount of time to meet a particular sales deadline. In addition, until the sale is complete, you will probably not be allocated the funds or time for a full-scale design.

Similarly, software architects may need to create a prototype. This may be for a project proposal, but it could also be to test out a new technology or framework, to create a **proof of concept** (**POC**) for some solution to a particular problem, or to explore how a certain quality attribute might be effectively met. As with project proposals, if the design objective is to build a prototype, the focus and scope of the software architecture design will be different from one that is being done for development.

It is important to keep the design objectives in mind as an architectural driver when software architecture design is about to begin.

Primary functional requirements

Another important type of input into the architecture design is the **primary functional requirements** that need to be satisfied. Primary functional requirements are those that are critical to the organization's business goals. In `Chapter 3`, *Understanding the Domain*, we discussed core domains, which refer to the part of the domain that makes the software worth writing. Some of the primary functionality will come from the core domain. It is what differentiates the organization from competitors.

Although satisfying functional requirements is a goal of software architecture design, keep in mind that not all functionality is affected by the architecture. While some functionality is highly affected by the architecture, other functionality can be delivered equally as well with different architectures.

Even in cases where functionality is not influenced by the architecture directly, functional requirements may be an architectural driver for other reasons. One example of this would be the need to make modifications to the functionality later. Maintainability and modifiability of the software are affected by the software architecture.

Quality attribute scenarios

Quality attributes are measurable properties of a software system. They are the *ilities*, such as maintainability, usability, testability, and interoperability. We have been stressing the importance of quality attributes since they play such an important part in the success of software systems, and because software architecture decisions will affect them.

This makes quality attributes one of the main architectural drivers for software architecture design. The design decisions that are made will determine what quality attributes will be met. As an architectural driver, quality attributes are typically described in the context of a particular scenario.

A **quality attribute scenario** is a short description of how the software system should respond to a particular stimulus. Scenarios make quality attributes measurable and testable. For example, a quality attribute of *performance* or a requirement that states a particular function should be fast is not measurable or testable. An actual example of a valid quality attribute related to performance would be as follows: *When the user selects the Login option, a Login page is displayed within two seconds*.

Prioritizing quality attribute scenarios

Prior to the start of the architecture design process, the quality attribute scenarios should be prioritized. It is helpful to be aware of the priority of each quality attribute scenario when designing the architecture. In addition to being able to plan accordingly, such as focusing on higher priority quality attributes first, there may be trade-offs involved when enabling certain quality attributes. Understanding the priorities will help you make better design decisions regarding the quality attributes and any trade-offs that need to be made.

Quality attribute scenarios can be prioritized by ranking them based on two criteria: their business importance and the technical risk associated with the scenario. A ranking scale of **High** (**H**), **Medium** (**M**), and **Low** (**L**) can be used.

Stakeholders can help to provide the ranking based on business importance, while the software architect typically provides the ranking based on technical risk. Once the rankings are complete, each quality attribute should have a combination of the two rankings.

If each quality attribute scenario were assigned a unique number, they could be placed in a table such as the following:

Business importance/technical risk	L	M	H
L	6, 21	7, 13	15
M	3, 10, 11	14, 16, 17	1, 5
H	4, 18, 19, 20	2, 12	8, 9

Quality attribute scenarios located toward the bottom-right side of the table will be of higher importance. The most important ones will be those with an *H*, *H* ranking, indicating they were ranked high on both criteria. Initial design iterations can focus on those scenarios first. Subsequent iterations can consider the most important quality attribute scenarios that remain, such as *H*, *M* and *M*, *H*, until all of the quality attribute scenarios have been considered.

Constraints

Constraints are decisions imposed on a software project that must be satisfied by the architecture. They typically cannot be changed. They can affect the software architecture design and are therefore an architectural driver.

Constraints are generally fixed from the beginning of the project and might be technical or non-technical. Examples of technical constraints include being required to use a specific technology, having the ability to deploy to a particular type of target environment, or using a specific programming language. Examples of non-technical constraints are being required to abide by a certain regulation, or that the project must meet a particular deadline.

Constraints may also be classified by whether they are internal or external. Internal constraints originate from within the organization and you may have some control over them. In contrast, external constraints come from outside of the business and you may not have any control over them.

Like the other architectural drivers, constraints need to be considered in the design as an input into the design process.

Architectural concerns

Architectural concerns are interests of the software architect that impact the software architecture. As a result, they are an architectural driver. Just as functional requirements and quality attributes are design issues important to stakeholders, architectural concerns are design issues important to the software architect.

Architectural concerns need to be considered part of the design, but are not captured as functional requirements. In some cases, they may be captured as quality attributes rather than architectural concerns, or an architectural concern may lead to new quality attribute scenarios that need to be met.

For example, a software architect may have concerns related to software instrumentation or logging. If not already recorded as part of a quality attribute, such as maintainability, the architectural concern may lead to a new quality attribute.

Good software architects will be able to recognize possible architectural concerns based on the type of software they are designing. Architectural concerns may also arise from previous architecture design iterations, so be aware that architecture changes may lead to new concerns being created from them.

Leveraging design principles and existing solutions

Designing a software architecture from scratch for a project with some level of complexity can be a challenging task. However, software architects have a number of tools at their disposal when designing an architecture.

The design issues facing a project may have already been solved by others, and rather than *reinventing the wheel*, those solutions can be leveraged in your architecture design. These architecture design principles and solutions, which are sometimes referred to as **design concepts**, are building blocks used to design a software architecture.

Selecting a design concept

There are many design concepts, so a software architect needs to know which ones are suitable for a particular problem, and then select the one that is most appropriate among the alternatives. You may also find cases where you need to combine multiple design concepts to create a solution.

Depending on the stage of the architecture design you are in and the nature of the problem, certain design concepts will make more sense than others. For example, a reference architecture would be useful when creating the initial structure of the architecture, but later in the design, when considering a specific quality attribute scenario, a tactic might be used.

Software architects generally determine which design concepts are available by using their knowledge and experience, leveraging the knowledge and experience of their teammates, and following best practices.

When choosing a specific design concept among multiple alternatives that have been identified, you'll want to weigh the pros and cons, as well as the cost of each alternative. Keep any project constraints in mind when selecting design concepts, as a constraint may prevent you from using certain alternatives.

Some of the design concepts available to you include software architecture patterns, reference architectures, tactics, and externally developed software.

Software architecture patterns

When designing a software architecture, some of the design issues that you will face have already been solved by others. **Software architecture patterns** provide solutions for recurring architecture design problems. Patterns are discovered while observing what people were doing successfully to solve a particular problem, and then documenting those patterns so that they can be reused. They can be leveraged in an architecture design if the software application has the same design issue.

Software architects should take advantage of the work and experience of others when they have a problem that can be solved by a pattern. The challenging part is to be aware of what patterns are available, and which ones are applicable to the problem you are trying to solve.

As with any design pattern though, you shouldn't try to force the use of one. You should only use an architecture pattern if it truly solves the design issue that you have and if it is the best solution given your context.

We will be exploring software architecture patterns in more detail in `Chapter 7`, *Software Architecture Patterns*.

Reference architectures

A **reference architecture** is a template for an architecture that is best suited to a particular domain. It is composed of design artifacts for a software architecture that provides recommended structures, elements, and the relationships between the elements.

Benefits of reference architectures

A reference architecture can answer many of the most common questions for systems that need a particular design. They can be very helpful to software architects because they provide a tested solution to a problem domain, and reduce some of the complexities involved in designing a software architecture. Reference architectures are proven, in both technical as well as business contexts, as viable solutions for certain problems.

Using a reference architecture allows the team to deliver a solution quicker, and with fewer errors. Re-using an architecture provides advantages such as quicker delivery of a solution, reduced design effort, reduced costs, and increased quality.

Leveraging the experiences of past software applications and learning from them can be of great value to software architects. They help us to avoid making certain mistakes and can prevent costly delays that may result from not using a previously proven approach.

Refactoring a reference architecture for your needs

Just as a design without using a reference architecture may require multiple iterations to achieve the final result, it is also the case when using a reference architecture. Design decisions will need to be made regarding the reference architecture.

During iterations for the architecture design, refactoring can take place on a reference architecture to meet the specific needs of the software application being designed. The amount of refactoring necessary depends on how closely the reference architecture meets the functional and quality attribute requirements.

Reference architectures may be created at different levels of abstraction. If you want to use one and it is not at the level of abstraction you need, you might still be able to learn from it, and use it as a guide when designing your own architecture.

For well-understood domains, there may be a number of reference architectures available to you. In contrast, if you are designing a solution for a greenfield system that is in a novel domain, there may be few, if any, available for you to use. Even for those types of projects though, you may find a reference architecture you can leverage, even if it is just for a portion of the design. It might just require more refinement and refactoring than when a more fitting reference architecture is available.

When you use a reference architecture, you adopt issues from that reference architecture that you will need to address. If a reference architecture deals with a particular design issue, you will need to make design decisions about that issue, even if you don't have a specific requirement related to it. The decision might very well be to exclude something from your architecture that is in the reference architecture.

For example, if a reference architecture includes instrumentation as a cross-cutting concern, you will need to make design decisions about instrumentation during your design.

Creating your own reference architecture

Once an organization has a completed software architecture, which may or may not have used a reference architecture, it can then become a reference architecture itself. When an organization needs to create new software applications, perhaps as part of a software product line, it can use the architecture of an existing product as a reference architecture.

Using a reference architecture from your own organization is just like using a reference architecture from somewhere else. You will reap benefits by doing so, but some amount of refactoring may be required to use it in a particular application. The added advantage is that the reference architecture would be likely to already be suited to your particular domain.

Tactics

Tactics are proven techniques to influence quality attribute scenarios. They focus on a single quality attribute, so they are simpler than other design concepts, such as architecture patterns and reference architectures, which aim to solve a greater number of design issues.

Tactics provide options to satisfy quality attributes, and the use of other design concepts, such as architecture patterns or an externally built framework, along with code, are required to fully complete the tactic.

We went over some tactics when we explored quality attributes, such as:

- Satisfying a maintainability quality attribute scenario by reducing complexity in a component by increasing cohesion and reducing coupling
- Increasing usability in a scenario by providing friendly and informative messages to the user
- Implementing a retry strategy in a process to handle a possible transient fault in order to improve an availability quality attribute scenario
- Satisfying a portability quality attribute scenario by increasing installability by ensuring that a software update process to a newer version properly cleans up the older version

Externally developed software

When designing a software architecture, you will be making design decisions for a number of design issues. Some of these design issues already have solutions in the form of concrete implementations that have been developed externally. Rather than build a solution in-house to solve a particular design issue, you can leverage software that has already been developed outside of the organization.

The **externally developed software** can come in different forms, such as a component, an application framework, a software product, or a platform. There are many examples of externally developed software, such as a logging library for logging functionality, a UI framework for creating user interfaces, or a development platform for server-side logic.

Buy or build?

One of the decisions that software architects need to make is the classic *buy or build* dilemma. When you are in need of a solution to a particular design issue, you will need to decide whether to buy or build it. When using the term *buy*, we are referring to using something built externally, and not necessarily the fact that it may have a monetary cost. Depending on what type of solution you are looking for, there may be a number of free solutions available to you, including those that are open source.

When deciding whether to use an externally developed solution or to build it in-house, you must first make sure that you understand the problem that you are trying to solve, and the scope of that problem. You will need to research whether externally developed software exists that will solve the design problem. If the problem is unique to your organization, there may not be any suitable software available.

You should also know whether or not the organization has, or can attain, resources to build, maintain, and support a solution. If the solution is to be built internally by the project team, there must be sufficient resources, including time, to build it.

Advantages/disadvantages of building

An advantage of building it internally is that the solution will be unique to your organization, and tailored to it. The organization will have complete control over the solution, including full ownership of the source code. This will allow the organization to modify it in any way that it wants. If a need arises to make changes or add functionality, the organization will be able to do so with full authority.

Another benefit of building it yourself is that there could be a competitive advantage. If the solution provides some feature that competitors do not currently have, building it and owning it could provide a strategic advantage to the organization.

The disadvantages of building it yourself are that it will require time and resources to do so. The end result may not have as robust a set of features as an externally developed solution. For example, if you are in need of a distributed, scalable, enterprise-level, full-text search engine as part of your application, it is probably impractical to build it yourself rather than use a proven solution that already exists.

Advantages/disadvantages of buying

Using an externally developed solution has its own set of advantages. It will save time, as no effort will need to be spent developing it. It may be of higher quality, assuming that it has already been tested and used in production. Feedback from other users of the software may have exposed problems that have already been fixed.

The external solution might be continually improved to achieve higher levels of quality and to introduce new features. Support and training may be available that your team will be able to leverage.

However, there are downsides to using an externally developed solution. There may be a cost to using such a solution. Depending on the license type, you may not have access to the source code and may be limited in how the solution can be used. If you can't modify the solution, then the solution's functionality will be controlled by someone else, and it may not exactly fit your needs. In addition, if there are issues with the solution, or you need it changed in some way, you will need to rely on an external organization.

Researching external software

In order to find out whether external software exists that will be a suitable solution for the problem being solved, or in order to select an external solution from multiple alternatives that might be available, some research will be required.

The software architect should consider the following:

- Does it solve the design problem?
- Is the cost of the software acceptable?
- Is the type of license that comes with the software compatible with the project's needs?
- Is the software easy to use? Does the team have resources that can use it?
- Can the software be integrated with the other technologies that are going to be used on the project?
- Is the software mature, providing stable releases?
- Does it provide the level of support that might be needed, whether that support is paid support or through a development community?
- Is the software widely known, such that the organization can easily hire resources familiar with it?

Creating one or more prototypes that use the possible candidate solutions is a good way to evaluate and compare them. A POC to ensure that it is a workable solution is a wise idea.

Should I use open source software (OSS)?

When searching for an externally developed solution that will solve a design problem, one possibility is to find **open source software** (**OSS**) that will fulfill your needs. OSS is written by the community and is intended for use by the community.

Given the wide availability and range of open source software, there are many solutions available for a variety of problems. It is much more common now to use open source solutions as part of a software application. Some organizations do not permit the use of open source software but if your organization does, then you should give it consideration as a viable alternative for a given task.

One consideration when selecting open source software is the license that is associated with it. The license dictates the terms and conditions under which the software can be used, modified, and shared. One set of open source licenses that are popular is a group of licenses that have been approved by the **Open Source Initiative** (**OSI**). Some of the OSI-approved licenses include (in alphabetical order):

- Apache License 2.0
- BSD 2-clause *Simplified* or *FreeBSD* license
- BSD 3-clause *New* or *Revised* license
- Common Development and Distribution License
- Eclipse Public License
- GNU General Public License (GPL)
- GNU Lesser General Public License (LGPL)
- MIT license
- Mozilla Public License 2.0

There are differences in the terms and conditions of the various licenses. For example, your application can incorporate open source software that uses the MIT license and you will be able to distribute your application without making it open source.

In contrast, if your application incorporates software that uses the GNU General Public License and you then distribute your application, your application would need to be made open source. This is true even if your application is free and you do not change the open source software you are using in any way. If your software is for internal use only and it is not distributed, then your application could remain proprietary and closed source.

Advantages of using open source software

There are benefits to using open source software, which explains its popularity. Using an open source solution for a design problem provides many of the same advantages as using one that has been purchased. You don't have to spend time building the solution, it may have a robust set of features, and it may already be a tested and proven solution with many other users.

Unlike software that must be purchased, open source software is freely available so there are cost savings. You just have to keep in mind the license that comes with the software.

If the open source software is a popular solution with an active community, it might be continuously improved with bug fixes and new features. You will be able to take advantage of this work. Bugs may be detected and fixed quickly because many people are using and working on the code. This is the idea behind *Linus's Law*, which is named after Linus Torvalds, the creator of the Linux kernel. Linus's Law basically states that given enough eyeballs, or people looking at the code, all bugs are shallow. In other words, with many people looking at the source code, problems will be detected sooner rather than later, and someone will be able to provide a fix.

Although some view open source software as less secure due to the availability of the code, some people see it as more secure because there are *many eyes* that are using, looking at, and fixing the code.

Another advantage of open source software is the fact that you have access to the source code. If necessary, your development team will be able to modify it just as you would with an in-house solution.

Disadvantages of using open source software

Despite its advantages, there are some disadvantages to using open source software that you should consider. Even though the software is free, there are still costs related to using it. Someone has to spend time integrating the solution into the software system, and there is an associated cost for that effort. If the open source software has to be modified in any way to suit the needs of the project, there is a cost related to that work as well.

If there is no one on the team who knows how to use the software, and it is complex enough that some training is required, learning how to use a piece of open source software may also take time.

Even for a popular open source project with an active community, there is no guarantee that the software will continue to experience support. There is always a risk of the software going out of favor. If the project is abandoned, you won't be able to rely on support for bug fixes or new features unless the development team performs that work for themselves.

One reason an open source software project may become less secure is if no one is actively working on it. Even if the project has not been abandoned, no one is necessarily reading the code. The average programmer writes much more code than they read. The existence of some prominent security bugs has shown that it is possible for critical security vulnerabilities to go undetected for some time.

Despite Linus's Law, the fact that the source code is readily available introduces a degree of security risk. Malicious individuals can analyze the source code to identify security vulnerabilities and attempt to take advantage of them.

Documenting the software architecture design

An important part of architecture design is documenting the design, including the many design decisions that are made during the process. This typically comes in the form of sketching architecture views and documenting the design rationale.

Sketching the architecture design

Software architectures are commonly documented through the creation of architecture views. Architecture views are representations of a software architecture that are used to document it and communicate it to various stakeholders. Multiple views of an architecture are typically required, as a software architecture is too complex to be represented in a single, comprehensive model.

Formal documentation of a software architecture through views will be covered `Chapter 12`, *Documenting and Reviewing Software Architectures*, and is not typically done as part of the design process. While that type of documentation comes afterward, informal documentation, in the form of sketches, should take place during the architecture design. Sketches can record the structures, elements, relationships between the elements, and the design concepts used.

The sketches do not necessarily need to use any formal notation, but they should be clear. While it's not necessary to sketch everything, at a minimum, you will want to sketch out important decisions and design elements. These sketches can be done on a whiteboard, on paper, or using a modeling tool.

Documenting the design by creating sketches during the design process will help you to create architecture views later. If you already have informal sketches, when the time comes to create formal documentation, you will find it to be an easier task.

Documenting the design as it occurs will also ensure that you do not forget any design details when it comes to creating the architecture views. Your architecture will be analyzed and validated later to ensure that it satisfies functional requirements and quality attribute scenarios, so it is helpful to record details during design that can then be used to explain how requirements and quality attributes were satisfied by the architecture design.

If you aren't able to sketch out a part of your design, you'll have to consider the possible reasons for that. Perhaps it is not well understood, too complex, you haven't put enough thought into how to communicate it, or there may be parts of it that are unclear to you. If that is the case, you should revisit the design until you are able to sketch it. If you can sketch the design created in an iteration effortlessly and clearly, your audience will be able to understand it.

Documenting the design rationale

Software architecture design involves making many design decisions, and software architects should document those decisions along with their design rationale. While design sketches may explain what was designed, they don't give any indication as to the design rationale.

A **design rationale** explains the reasons, and justification, behind the decisions that are made during the design of the software architecture. Design rationale can also include documentation on what decisions were *not* made, as well as alternatives that were considered for decisions that were made. Reasons for rejection can be recorded for each alternative that was not selected.

Recording design rationale can be useful during the design process, as well as once the design is complete. Software architects who document their design rationale are afforded an opportunity to clarify their thoughts and arguments as they capture the design rationale, which may even expose flaws in their thinking.

Once the design rationale is documented, anyone who wants to know why a particular design decision was made, even after some time has passed, can refer to it. Even individuals who were involved with a design decision, including the software architect, may forget the rationale behind a particular decision and will be able to refer to the documentation.

The design rationale should refer to the specific structures that were designed and the specific requirements that they intended to meet. Some software design tools provide functionality that can assist the software architect in capturing the design rationale.

A complete design rationale provides a history of the software architecture design process. There are a number of uses for design rationale, such as for design evaluation, verification, knowledge transfer, communication, maintenance, documentation, and reuse.

Design rationale for design evaluation

Design rationale can be used to evaluate different software architecture designs and their design choices. The various designs can be compared with each other, and an understanding can be gained as to the situations in which one design would be chosen over another.

Design rationale for design verification

The purpose of software architecture design verification is to ensure that the software system, as designed, is the software system that was intended. It verifies that the software architecture meets the requirements, including the quality attributes, and works as expected. The design rationale can be used as part of the verification.

Design rationale for design knowledge transfer

The design rationale can be used for knowledge transfer to team members, including those who may join the team later, either during development or after the software goes into its maintenance phase.

Team members can learn about the design decisions and the reasons behind them by reviewing the design rationale. It is particularly useful for knowledge transfer when the original software architect, and others who collaborated on the design of the software architecture, are no longer available to provide the information in other ways.

Design rationale for design communication

It will be necessary at different times to communicate the design of the software architecture to various stakeholders and other individuals. The information provided by the design rationale adds value to the overall communication.

In addition, the design rationale can be used by those who are reviewing the software architecture so that they can learn the reasons behind particular design decisions.

Design rationale for design maintenance

During the maintenance phase of a software project, it is helpful to know the design rationale for the decisions that went into the software architecture design. When a certain piece of the software needs to be changed for maintenance, the design rationale can assist in determining what areas of the software will require modifications.

It can also be used to identify weaknesses in the software, and areas of the software that could be improved. For example, based on certain design decisions, quality attributes may be enabled or inhibited, and if changes are being considered that would alter those decisions, team members could be aware of the reasons behind those decisions.

The design rationale will also point out design alternatives that were not chosen, allowing those considering modifications to either avoid previously rejected design alternatives or to at least be knowledgeable about the reasons those alternatives were rejected in order to make an educated decision.

Design rationale for design documentation

The software architecture must be documented, and the design rationale is an important part of that documentation. If the documentation only shows the design, those looking at it will know *what* was designed but won't know *why* it was designed that way. They also won't be aware of the alternatives that were considered, and why those alternatives were rejected.

Design rationale for design reuse

Software architecture reuse involves creating multiple software applications using *core assets*, allowing architectural components to be reused across multiple software products. Organizations seek to take advantage of efficiencies that can be gained when reusing architectural components to build multiple software products as part of a *software product line*.

Capturing design rationale can facilitate successful architectural reuse. It can help designers understand what parts of the application can be reused. It may also provide some insight into where modifications can be made to a component in order to reuse it in the application. Due to the variation between software products, reusable components are typically designed with *variation points*, or places where modifications can be made in order to adapt the component for use in a particular software product. Understanding the design rationale will help designers use the component properly, and prevent harmful modifications from being made.

Using a systematic approach to software architecture design

If you are going to dedicate some time to designing the architecture of a software system, and not just let it *emerge* after implementing features, you should do so in a systematic way.

Software architects need to ensure that the architecture they are designing will satisfy the architectural drivers, and a systematic approach can assist in accomplishing that goal. In *Designing Software Architectures, A Practical Approach*, the following is said about using an architecture design process:

> *"The question is, how do you actually perform design? Performing design to ensure that the drivers are satisfied requires a principled method. By "principled", we refer to a method that takes into account all of the relevant aspects that are needed to produce an adequate design. Such a method provides guidance that is necessary to guarantee that your drivers are satisfied."*

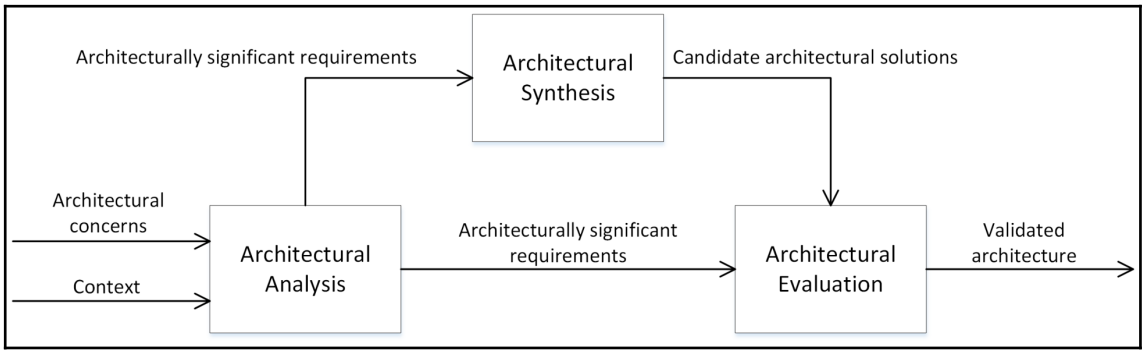
Using an established architecture design process will provide you, as the software architect, with guidance on how to go about designing an architecture that will satisfy functional requirements and quality attribute scenarios. There are a number of design processes that can be used for software architecture. Although they differ from each other, including differences in terminology, they also have some fundamental commonalities.

A general model of software architecture design

The paper, *A general model of software architecture design derived from five industrial approaches*, by Christine Hofmeister, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, and Pierre America, compared five different architecture design methods to come up with a general model of architecture design based on the similarities. Having a general model helps us to understand the types of activities that are typically performed in a software architecture design process and allows us to compare the strengths and weaknesses of different processes.

It was found that most architecture design processes involve analyzing architectural drivers, designing candidate solutions that will satisfy the architectural drivers, and then evaluating the design decisions and candidate solutions to ensure that they are correct.

The three main design activities that were identified in *A general model of software architecture design derived from five industrial approaches* are **Architectural Analysis**, **Architectural Synthesis**, and **Architectural Evaluation**:



Architectural analysis

During **architectural analysis**, the problems that the architecture is trying to solve are identified. These are sometimes referred to as **architecturally significant requirements** (**ASRs**) because they will influence the design of the architecture. Not all of the design issues that must be considered are requirements though. We must address all of the architectural drivers, which include design objectives, primary functional requirements, quality attribute scenarios, constraints, and architectural concerns.

The output of this activity is a set of architectural drivers that will serve as the input to architectural synthesis.

Architectural synthesis

The **architectural synthesis** activity is where solutions are designed on the basis of the set of architectural drivers that were identified in the architectural analysis activity. It is during this activity that we leverage design concepts such as architecture patterns, reference architectures, tactics, and externally developed software, and combine them with the design of structures, elements, and relationships between the elements. This produces solutions for the set of architectural drivers.

The output of this activity is one or more candidate solutions for the problems selected.

Architectural evaluation

In the **architectural evaluation**, the candidate solutions that were designed during architectural synthesis are evaluated to ensure that they solve the problems that they were intended for, and that all of the design decisions that were made are correct.

At the conclusion of the architectural evaluation activity, each candidate solution has either been validated or invalidated. We will cover reviewing software architectures in `Chapter 12`, *Documenting and Reviewing Software Architectures*.

Architecture design is an iterative process

Another important similarity found in architecture design is the fact that it is an iterative process. Designing a software architecture is too complex to address all of the architectural drivers simultaneously.

The design of the architecture occurs over multiple iterations until all architectural drivers have been addressed. Each iteration starts by selecting the architectural drivers that will be considered for that iteration. If candidate solutions have been validated after they have been evaluated, those design decisions are integrated into the overall architecture.

If there are no more architectural drivers that need solutions, the validated architecture is complete. If outstanding architectural drivers exist, a new iteration will begin.

Selecting an architecture design process

Now that we understand the fundamental activities that occur during software architecture design, which one do we use? There are many different software architecture design processes. One way that we can compare design processes is to examine the activities and artifacts of the design process:

- What are the activities and artifacts of the design process?
- Are there any activities/artifacts that you think are not needed?
- Are there any activities/artifacts that you feel are lacking?
- What are the techniques and tools of the design process?

If it helps, you can compare the activities and artifacts of the design process with those that exist in the general model. Some of the activities and artifacts of the design process may have corresponding ones in the general model, although different names may be used. There may also be activities and artifacts in the design process that do not have any corresponding ones in the general model, as well as ones in the general model that do not exist in the design process.

After doing this type of comparison, you should have an understanding as to what each design process entails, along with their strengths and weaknesses. This knowledge can be used to select the one that is most suited to your project.

The software architect can modify a design process to make it more suitable for a project's needs, although such changes should only be done thoughtfully. If, after analyzing and selecting a design process, you feel that a particular activity or artifact is not needed, you could remove it.

Conversely, if you see that a design process lacks an activity and/or artifact, you can change the process to include it. You might be able to draw on a technique, tool, or even another design process to supplement what you feel is missing from the design process you want to use.

So far, we have been discussing architecture design processes in fairly general terms, so now let's explore several concrete ones at a high level. Three of the architecture design processes available to you are ADD, Microsoft's technique for architecture and design, and the ACDM.

Attribute-driven design (ADD)

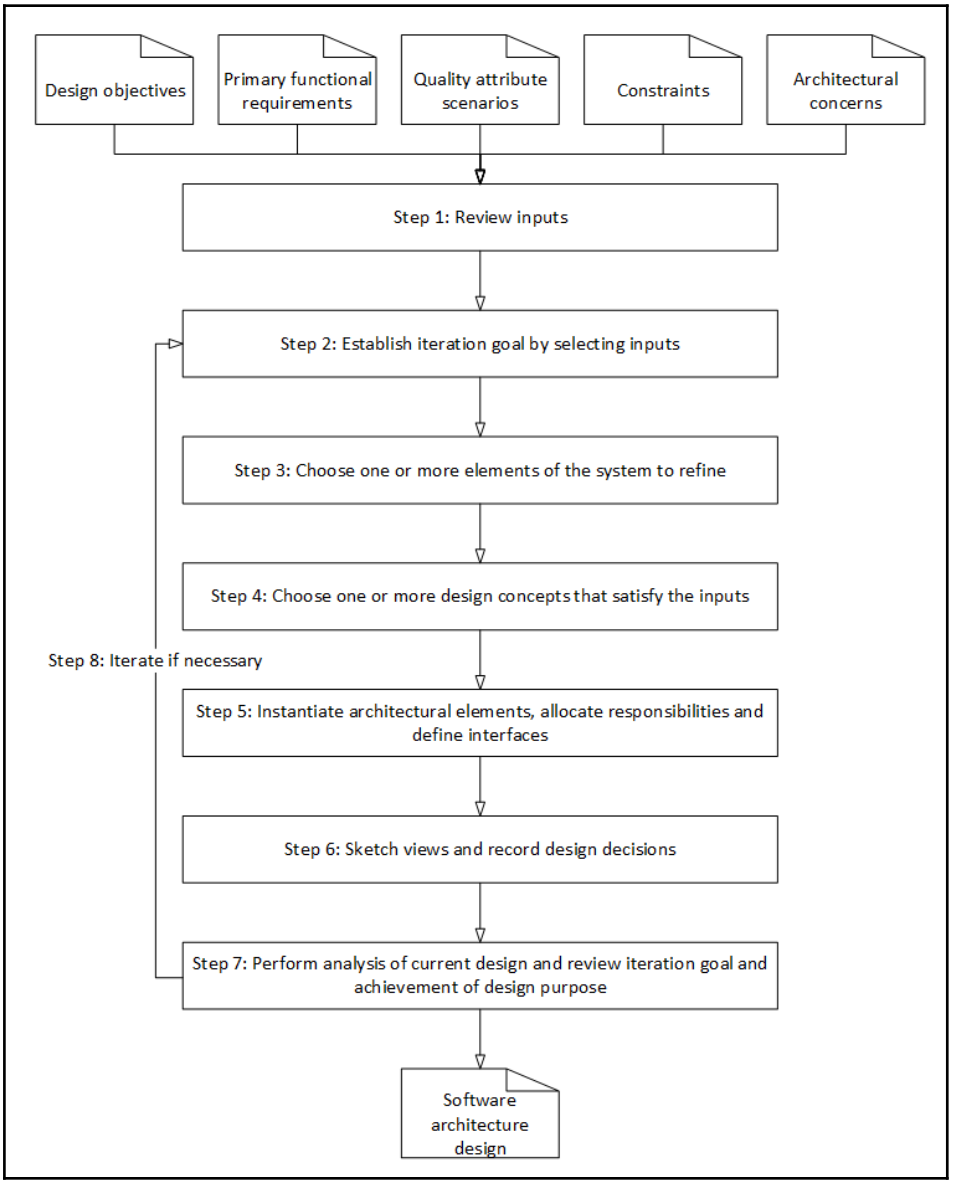
Attribute-driven design (**ADD**) is one of the systematic approaches for designing software architectures. It is an iterative, organized, step-by-step method that can be followed during architectural design iterations.

This method pays particular attention to software quality attributes during the design process. As a result, one of the primary benefits of using ADD is that you begin to consider quality attributes early in the design process.

Enabling a quality attribute in a software architecture design may affect other quality attributes. Consequently, trade-offs between quality attributes may be necessary. By focusing on quality attributes using the ADD method, these types of trade-offs can be considered at an early stage during the process.

The ADD process is specifically focused on architecture design and, as such, doesn't cover the entire architectural life cycle. The process doesn't include the gathering of architectural drivers, documenting the architecture, or evaluating the architecture once it is designed. However, you can combine ADD with other methods to fill in these gaps.

ADD is a widely-used method for software architecture design, and has been used successfully on a variety of software applications. There are eight steps in the attribute-driven design process:



Step 1 – Reviewing inputs

The first step in the ADD process is to review the inputs into the attribute-driven design. Before the design starts, we want to ensure that we are clear on the overall design problem we are solving.

The inputs are the architectural drivers that we reviewed earlier:

- Design objectives
- Primary functional requirements
- Quality attribute scenarios
- Constraints
- Architectural concerns

If the software system is either a brownfield system or it is not the initial iteration for the architecture design of a greenfield system, there is at least some part of an architecture already in place. This existing architecture must be considered as part of the input into the iteration.

Step 2 – Establishing the iteration goal and selecting inputs to be considered in the iteration

Once inputs are reviewed, one or more design iterations will take place, with each iteration beginning with *Step 2*. If you are using an agile methodology, multiple iterations will take place until the architecture is complete and the design purpose has been accomplished. An agile methodology is preferred, and is more common, as attempting to provide solutions for all of the architectural drivers at once can be too difficult.

At the start of each iteration, we want to establish the design goal for that iteration. We should be able to answer the question, *What design issue are we trying to solve in the iteration?* Each goal will be associated with one or more inputs. The inputs, or architectural drivers, that are relevant to the goal are identified and will be the focus of the iteration.

Step 3 – Choosing one or more elements of the system to refine

Based on the iteration goal and the architectural drivers that we want to create a solution for, we must select the various elements that we want to decompose.

If your project is a greenfield system and this is the first iteration, you begin at the highest level and start by decomposing the system itself. For any other iteration, the system has already been decomposed to some degree. You would select one or more of the existing elements to focus on for this iteration.

Step 4 – Choosing one or more design concepts that satisfy the inputs considered in the iteration

Once elements have been selected for decomposition, we need to select one or more design concepts that can be used to meet the iteration goal and satisfy the inputs (architectural drivers). Design concepts refer to design principles and solutions such as architecture patterns, reference architectures, tactics, and externally developed software.

Step 5 – Instantiating architectural elements, allocating responsibilities, and defining interfaces

Based on the design concepts that can be leveraged for this iteration, analysis is performed so that details can be provided regarding the responsibilities for the elements being decomposed, along with the public interfaces of those elements that will be exposed.

Each element being decomposed (parent element) may yield one or more child elements. By considering the responsibilities of the parent element, we can assign responsibilities to the various child elements. All of the responsibilities of the parent element are considered, whether or not they are architecturally significant.

Step 6 – Sketching views and recording design decisions

Views should be sketched recording the solution designed so that it can be communicated. In this step, all of the design decisions that were made during this particular iteration are documented. This documentation should also include the design rationale.

The artifacts created in this step can simply be sketches and do not have to be the formal, detailed software architecture views. In the ADD process, the creation of the architecture views comes later, but the design decisions made in this iteration should be reflected in sketches that can then be used in the formal architecture views later.

We will explore documenting software architectures in `Chapter 12`, *Documenting and Reviewing Software Architectures*.

Step 7 – Performing analysis of current design and reviewing the iteration goal and design objectives

In this final step of a software architecture design iteration, the software architect and other team members should analyze the current design. The design decisions are analyzed to ensure that they are correct and satisfy the iteration goal and architectural drivers that were established for the iteration.

The result of this analysis should determine whether more architecture design iterations will be necessary.

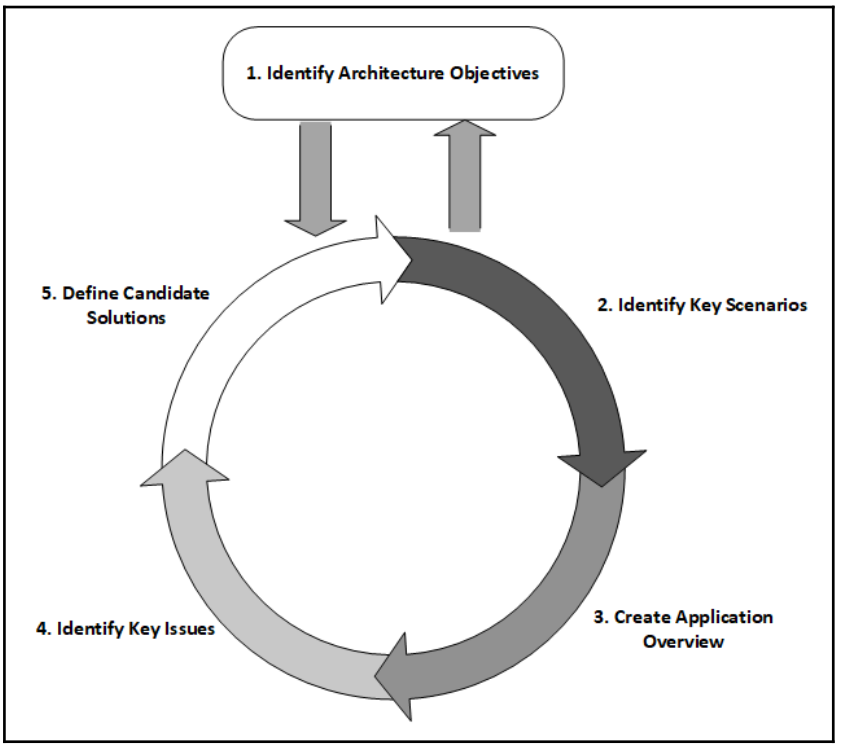
Step 8 – Iterating if necessary

If it is decided that more iterations are needed, the process should go back to *Step 2* for another iteration. As a software architect, there will be times where you feel more iterations are necessary, but something will prevent you from conducting more iterations. For example, the project management team may decide that there is not enough time for more iterations and that the architecture design process is done.

If no further iterations will take place, the software architecture design is complete.

Microsoft's technique for architecture and design

Another example of a systematic approach for designing software architectures is **Microsoft's technique for architecture and design**. Like ADD, it is an iterative, step-by-step method. This process can be used to design the initial architecture as well as to refine it later, if necessary. There are five steps in the process:



Step 1 – Identifying architecture objectives

The design process begins by identifying the objectives you want to achieve for the architecture. The purpose of this step is to ensure that the design process has clear objectives so that the solution focuses on the appropriate problems. Once design iterations start, we want to ensure that we are clear on the overall design problems that we are solving.

The various architectural drivers, such as design objectives, primary functional requirements, quality attribute scenarios, constraints, and architectural concerns all combine to form the architecture objectives. The software architect should also consider who will consume the architecture. The architecture design might be used by other architects, developers, testers, operations personnel, and management. Both the needs and the experience levels of the various people who will view and use the architecture design should be considered during the design process. As with ADD, if the software system has an existing architecture, either because it is a brownfield system or design iterations have already taken place, the existing architecture is another consideration.

Step 2 – Identifying key scenarios

Once the design objectives are identified and established, one or more design iterations will be necessary. *Step 1* will only occur once, and each design iteration will begin with *Step 2*. *Step 2* focuses on identifying key scenarios for the software application.

In this design process, a scenario is defined as a more encompassing user interaction with the software system, and not just a single use case. Key scenarios are the most important of these scenarios and are required for the success of the application. Scenarios could be an issue, an architecturally significant use case (one that is business critical and has a high impact), or involve an intersection between functional requirements and quality attributes. Once again, keep in mind that there may be trade-offs with quality attributes, so the scenarios should take those into consideration.

Step 3 – Creating application overview

In this step, armed with architecture objectives and key scenarios, an application overview is created. An application overview is what the architecture will look like when it is complete. An application overview is intended to connect an architecture design with real-world decisions.

Creating an application overview consists of determining your application type, identifying deployment constraints, identifying architecture design styles, and determining relevant technologies.

Determining your application type

The software architect must determine what type of application is appropriate based on the objectives and key scenarios. Examples of application types include web, mobile, service, and Windows desktop applications.

It is possible that a software application may be a combination of more than one type.

Identifying your deployment constraints

When designing a software architecture, among the many constraints that you may have to consider are constraints related to deployment. You may be required to follow particular policies of the organization. The infrastructure and target environment of the software application may be dictated by the organization, and such constraints may be something that you will have to work around.

The earlier that any conflicts and issues related to constraints can be identified regarding the software application and the target infrastructure, the easier those issues can be resolved.

Identifying important architecture design styles

Architecture design styles, also known as architecture patterns, are general solutions to common problems. Using an architecture style promotes reuse by leveraging a known solution to a recurring problem.

Identifying the appropriate architecture design style, or a combination of styles, that will be used in the software application is an important part of creating an application overview. We will go into detail about various architecture patterns in `Chapter 7`, *Software Architecture Patterns*, and `Chapter 8`, *Architecting Modern Applications*.

Determining relevant technologies

At this point, you are ready to select relevant technologies for your project. The decisions are based on the type of application, which was determined earlier, the architectural styles, and the key quality attributes.

In addition to technologies specific to the application type (for example, selection of a web server for a web application), technologies will be needed for categories such as application infrastructure, workflow, data access, database server, development tools, and integration.

Step 4 – Identifying key issues

This step of the process involves identifying the important issues you may be facing in the architecture. These issues may require additional focus because they are areas where mistakes are most likely to be made.

Key issues typically map in one form or another to either quality attributes or cross-cutting concerns. We took a look at quality attributes in `Chapter 4`, *Software Quality Attributes*, and will be exploring cross-cutting concerns in `Chapter 9`, *Cross-Cutting Concerns*.

Analyzing quality attributes and cross-cutting concerns closely based on the issues you identify will allow you to know which areas to give extra attention to in your design. The design decisions that are made as a result should be documented as part of the architecture.

Step 5 – Defining candidate solutions

Once key issues are identified, candidate solutions can be created. Depending on whether or not this is the first iteration, either an initial architecture is created, or the existing architecture is refined to include the solutions that were designed in the current iteration.

Once candidate solutions are integrated into the architecture design for the current iteration, the architecture can be reviewed and evaluated. We will go into further detail on reviewing software architectures in `Chapter 12`, *Documenting and Reviewing Software Architectures*.

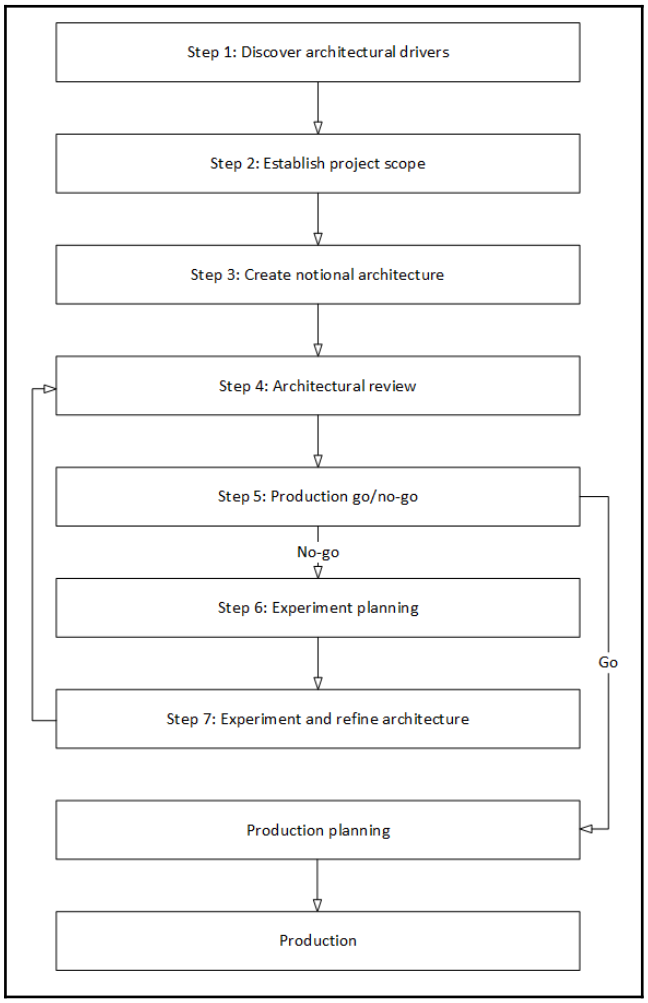
If it is determined that more work is necessary for the architecture design, a new iteration can begin. The process goes back to *Step 2* so that key scenarios can be identified for the next sprint.

Architecture-centric design method (ACDM)

The **architecture-centric design method** (**ACDM**) is an iterative process used to design software architectures. It is a lightweight method with a product focus and seeks to ensure that the software architecture maintains a balance between business and technical concerns. It attempts to make the software architecture the intersection between requirements and the solution.

Like all architecture design processes, the ACDM provides guidance to software architects as they design an architecture. While it covers the complete life cycle of software architecture, it is not a complete development process. It is designed to fit in with existing process frameworks though so that it can be used in conjunction with other methods to cover activities outside of architecture. It does not have to replace an existing process framework and can complement it instead.

There are some minor variations in the number and naming of the steps involved with the ACDM, but the process is essentially the same. Let's go over the ACDM, which is a seven-step process:



Step 1 – Discovering architectural drivers

The first step in the ACDM is to meet with stakeholders to determine the architectural drivers, which include design objectives, primary functional requirements, quality attribute scenarios, constraints, and architectural concerns. The prioritization of quality attribute scenarios also takes place in this step.

Step 2 – Establishing project scope

In this step, the architectural drivers established in *Step 1* are reviewed. First, consolidation of the information gathered takes place to remove duplicate architectural drivers.

Next, if any of the architectural drivers gathered are unclear, missing, or incomplete, additional information will be needed. The same is true of any requirements or quality attribute scenarios that are not measurable or testable.

If any clarification or additional information is needed, it will be gathered in this step from the relevant stakeholders.

Step 3 – Creating notional architecture

Using the architectural drivers, a notional architecture is created. It is the first attempt at designing the architecture. The initial representations of the structures that make up the architecture are created and documented.

Not a lot of time is typically spent on the notional architecture. The idea is that the architecture will be refined through multiple iterations until it is complete.

Step 4 – Architectural review

During this step, a review is conducted on the architecture as it currently exists. Reviews may be conducted internally, externally with stakeholders, or there may even be multiple review sessions so that both internal and external ones can be conducted.

The purpose of the review is to ensure that all of the design decisions are correct and to uncover any potential issues or problems with the architecture. For a given design decision, alternative approaches can be discussed, along with the trade-offs and the rationale behind the decision, in order to determine whether the best alternative was taken.

Step 5 – Production go/no-go

Once the architectural review is complete, a decision is made as to whether the architecture is complete and ready for production, or if further refinement is needed. In the ACDM context, *production* refers to implementation, including using the architecture in the detailed design of elements, coding, integration, and testing.

Any risks identified during the architectural review are considered in the production go/no-go decision. The decision does not have to be an all-or-nothing one. It is possible that only parts of the design require further refinement, in which case a portion of the design can move on to production.

If the production decision is a go, and no further refinements are needed, the process can skip ahead to production planning, and eventually on to production. However, if the production decision is a no-go, then the process moves on to *Step 6*.

Step 6 – Experiment planning

In this step, any experiments that the team feels are necessary are planned. The purpose of an experiment may be to resolve an issue uncovered during the architectural review, to gain a greater understanding of one or more architectural drivers, or to improve elements and modules of the design before they are committed to the overall architecture.

Experiment planning includes solidifying the goals of the experiment, estimating the level of effort, and assigning the resources that will be needed.

Step 7 – Experimenting with and refining the architecture

Any experiments that were planned are executed during this step. The results of the experiments are recorded. Based on the results of the experiment, if the architecture needs to be refined, it is done during this step.

After the refinement is complete, the process goes back to *Step 4* so that another architectural review can take place.

Production planning and production

Once architecture design iterations are complete, and the architecture is ready to move into production, production planning is conducted. Once again, *production* in the ACDM context refers to using the architecture in implementation.

Given this context, production planning involves planning the design and development of elements, scheduling the work, and assigning tasks to resources. The project management team creates plans for the work, and bases them, in part, on the architecture.

Once the architecture can be moved to production, it can be used by development teams for the detailed design of elements, coding, integration, and testing.

Architecture development method (ADM)

The **architecture development method** (**ADM**) is a step-by-step software architecture design approach specifically made for enterprise architectures. The ADM was created from the contributions of many software architecture practitioners.

Like the other architecture design methods that we have covered, the ADM is an iterative process. The process as a whole is iterative, but it is also iterative between phases and within a single phase. Each iteration is an opportunity to revisit scope, the level of detail to be defined, schedules, and milestones.

The Open Group Architecture Framework (TOGAF)

The ADM is a core part of **The Open Group Architecture Framework** (**TOGAF**), which is a framework for enterprise architecture. TOGAF provides a detailed method, the ADM, along with a set of tools for developing enterprise architectures.

TOGAF is maintained by *The Open Group*, which is a global industry consortium that focuses on using open and vendor-neutral technology standards to help organizations achieve business objectives.

TOGAF architecture domains

Four standard architecture areas, or architecture domains, for enterprise architecture are defined by TOGAF. They are business, data, applications, and technology architectures. These domains are sometimes referred to as the BDAT domains.

All four of these architecture domains are considered during the ADM and we will explore them in more detail when we go over the various phases of the ADM.

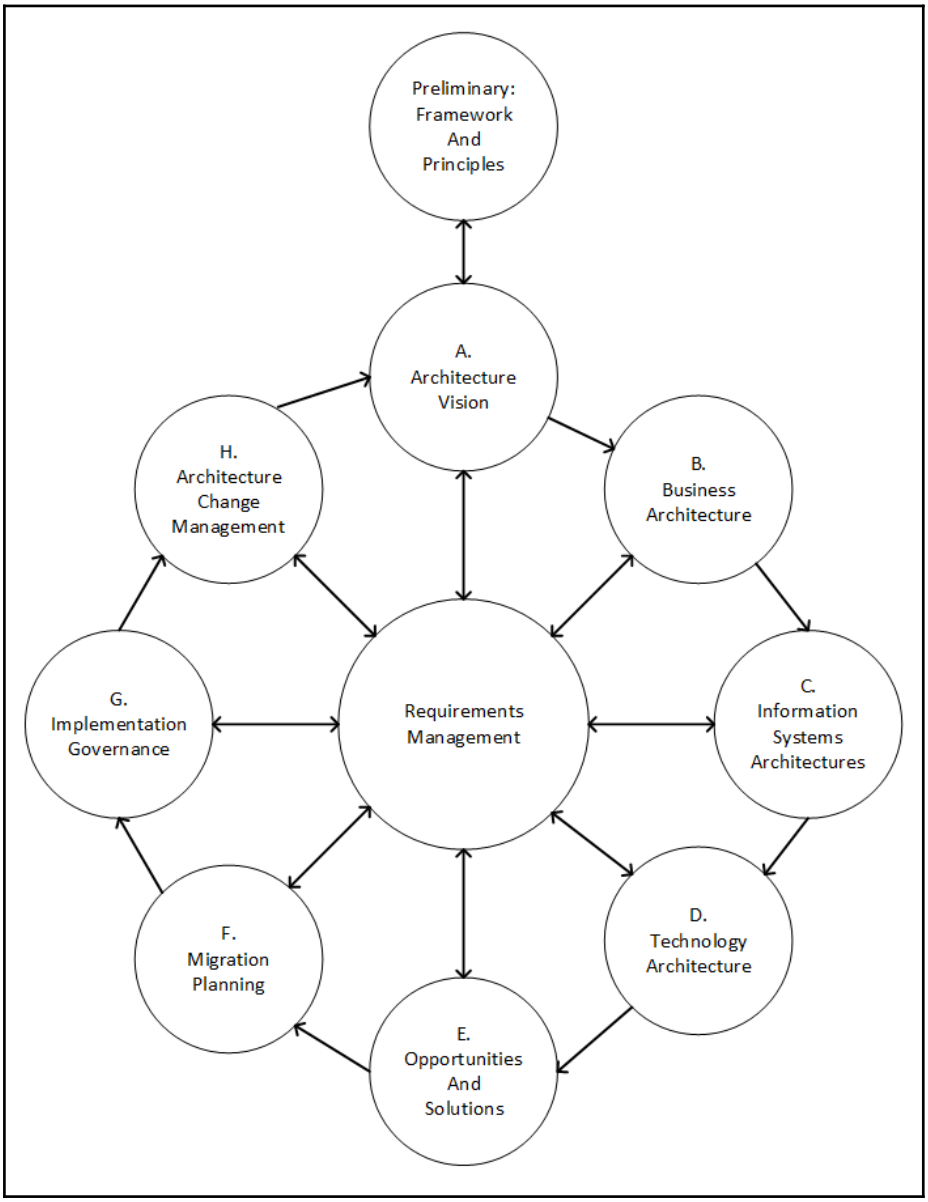
TOGAF documentation

The TOGAF documentation is broken up into the following seven sections:

- **Part I – Introduction**: The first part is an introduction to the concepts of enterprise architecture, the TOGAF approach, and definitions of relevant terms used in TOGAF.
- **Part II – Architecture development method**: This section details the **architecture development method** (**ADM**), which is the core of TOGAF. We will be focusing our attention on the ADM part of the TOGAF.
- **Part III – ADM guidelines and techniques**: This part of the documentation provides guidelines and techniques for applying TOGAF and the ADM.
- **Part IV – Architecture content framework**: In this part, information is provided on the TOGAF content framework, including the architectural artifacts and deliverables that are part of the process.
- **Part V – Enterprise continuum and tools**: This section covers the architecture repository for an enterprise, including the categorization and storage of architecture artifacts.
- **Part VI – TOGAF reference models**: In this section, various architectural reference models are provided, including the **TOGAF Foundation Architecture** and the **Integrated Information Infrastructure Reference Model** (**III-RM**).
- **Part VII – Architecture capability framework**: The final part provides guidelines on establishing and operating an enterprise architecture capability within an enterprise, including processes, skills, roles, and responsibilities.

Phases of the ADM

The architecture development method consists of multiple phases. There is a preliminary phase in which the organization prepares for a successful software architecture implementation. After this preliminary phase, there are eight phases to the process:



Each phase continuously checks with requirements to ensure that they are being met. Organizations can modify or extend the process to meet their needs, and it is usable with the deliverables of other frameworks if it is decided that those deliverables are more suitable.

Phase A – Architecture vision

In this step of the ADM, the team defines the overall vision for the enterprise architecture, including its capabilities and business value. The team agrees on items such as scope, business goals, business drivers, constraints, requirements, roles, responsibilities, and scheduling. These decisions are documented in the Statement of Architecture Work, which is a deliverable for this phase. The document typically contains the following:

- Architecture project request and background information
- Project description and scope of the architecture
- An overview of the architecture vision
- Change of scope procedures
- Roles, responsibilities, and deliverables for the project
- Details on the acceptance criteria and procedures
- Project plan and schedule

Phase B – Business architecture

Business architecture is one of the four architecture domains defined in TOGAF. The business architecture focuses on the business and/or service strategies of the organization, along with its business environment. An understanding of the business architecture is a prerequisite to perform architecture work on the other three domains defined in TOGAF (data, application, and technology).

The goal of this phase is to determine the target business architecture for how the enterprise achieves its business objectives and its strategic drivers. In order to create a roadmap on how to reach the target state, the following four steps are undertaken:

1. Gain an understanding of the current state of the architecture
2. Refine and validate the target state of the architecture
3. Determine the gap that exists between the current and target states of the architecture
4. Create a roadmap to transition between the current and target architecture states

Phase C – Information systems architectures

Data and application architecture are two of the other architecture domains defined in TOGAF. The data architecture focuses on an organization's data and how it is managed, while application architecture involves the enterprise's software applications.

The results from the architecture vision and business architecture phases are used to determine the architectural changes that will be necessary to an enterprise's data and application architectures.

As was the case with *Phase B*, the current and target states of the architecture are compared so as to determine the gap between the two. This allows for an architecture roadmap to be created for the candidate application and data components that will be needed to bridge the gap.

Phase D – Technology architecture

Technology architecture, one of the other architecture domains defined in TOGAF, involves the enterprise's infrastructure components. This includes the hardware and software necessary to support the enterprise's business, data, and application architectures.

The goal of this phase is to develop the target technology architecture that will support the enterprise's solutions. An assessment of the enterprise's current infrastructure capabilities is completed and compared with the desired target state so that the gap between them can be identified. From there, a roadmap of the target state for the technology architecture can be created along with the candidate components.

Phase E – Opportunities and solutions

This phase focuses on how to deliver the target architecture as we move from a conceptual view of the target architecture toward implementation. The roadmaps created in *Phase B*, *Phase C*, and *Phase D* are consolidated into an overall architecture roadmap. The candidate solutions that were created in the previous phases are organized into high-level candidate work packages.

The overall architecture roadmap, which includes all of the gaps between the current and target states of the architecture, is used to determine the best approach on how to deliver the target architecture.

If an incremental approach is to be taken, transition architectures are identified so that business value continues to be delivered.

Phase F – Migration planning

The overall architecture roadmap and the candidate work packages are used to plan the implementation of the architecture. The software architect works with the enterprise's project and program management teams to determine existing or new projects that can be used for the work.

An enterprise's existing processes for change and project management can be used to plan the necessary initiatives.

Phase G – Implementation governance

Implementation governance, along with the next phase, architecture change management, runs in parallel with the implementation of the architecture. The development of the architecture takes place using the enterprise's existing software development process.

This phase ensures that software architects stay engaged during implementation by assisting and reviewing the development work. Software architects must ensure that the architecture being implemented is achieving the architecture vision.

Phase H – Architecture change management

During implementation, issues may arise that require decisions to be made. It may be found that changes are necessary to the candidate solutions.

Software architects are involved with the enterprise's change management process to make decisions regarding proposed changes. Changes to the architecture must be managed and there must be a continuous focus on ensuring that the architecture meets requirements and stakeholder expectations.

Tracking the progress of the software architecture's design

During the software architecture design process, you will want to keep track of the design's progress. Keeping track of progress enables you to know how much of the design work is complete, and how much of it remains. The remaining work can be prioritized, assisting software architects in determining what should be worked on next. In addition to tracking progress, it serves as a reminder of design issues that are still outstanding, so that nothing is forgotten.

The technique that management will want to use to track progress really depends on your project, software development methodology, and the organization. If you are using an agile methodology such as Scrum, you may be using product and sprint backlogs to track progress.

Using a backlog to track the architecture design progress

A **product backlog** contains a complete list of the features and bugs of the product. You should consider creating a product backlog that is specific to the software architecture of the project, which is separate from the other product backlog. An architecture product backlog would contain items, design issues, design decisions that need to be made, and ideas that are specific to the architecture design.

Prior to each sprint, sprint planning takes place. The team selects items from the product backlog that they will work on and complete during the upcoming sprint. Once tasks can be created for the product backlog items, they can be assigned to a resource and then tracked for progress. Items from the product backlog that get selected for a sprint are then moved to the sprint backlog. Once an item is completed, it can be removed from the backlog.

Before sprint planning takes place though, the product backlog items should be prioritized, as the prioritization may affect what gets selected for a particular sprint.

Prioritizing the backlog

The list of features and bugs in the product backlog should be prioritized by the team, which assists with project planning so that teams know which items to focus on first.

Backlog prioritization is not something that occurs just once. As the architecture backlog changes, priorities may need to change as well. You can revisit the prioritization of architecture backlog items as many times as necessary.

Product backlog items should be linearly ordered based on criteria. One set of criteria that has been used in practice to prioritize backlog items is called the DIVE criteria.

DIVE criteria

DIVE is an acronym that stands for the types of criteria that are used to prioritize product backlog items. It focuses on **D**ependencies, to **I**nsure against risks, business **V**alue, and estimated **E**ffort as the factors used to determine priority.

Dependencies

Some product backlog items will be dependent on others, and therefore those dependencies will need to be completed first. For example, if item *A* depends on item *B*, *B* would be prioritized higher than item *A*.

Insure against risks

When prioritizing backlog items, you want to insure against risks, which include both business and technical risks. Taking potential risks into consideration may lead the team to prioritize a backlog item higher or lower when compared to other backlog items.

Business value

The business value of a product backlog item is an important criterion for prioritization. Product backlog items with greater levels of business value may be deemed a higher priority. The input of relevant stakeholders can help to determine the business value of a product backlog item.

Estimated effort

The estimated level of effort for a product backlog item may be a factor when prioritizing work. This may be due to factors such as scheduling or resource availability. There may be cases where a product backlog item has a large estimated effort, and the team wants to tackle the item sooner rather than later to ensure that it will be completed in time.

Active and dynamic architecture backlogs

As with any product backlog, the architecture backlog is not static and will evolve as the architecture design takes place. As architecture design iterations are completed, new architectural drivers may be uncovered, necessitating the need for new items to be added to the backlog.

Another reason that items may be added to the architecture backlog is when issues are discovered with the architecture. When the design is reviewed, a problem may become apparent, requiring further work to be done.

As architectural design decisions are made, it may cause the creation of new architecture backlog items. When a design decision is made, new concerns may arise from that decision. For example, if it is decided that the application will be a web application, backlog items related to security, session management, and performance that are specific to web applications may need to be added to the architecture backlog if they did not already exist. Changes to the architecture backlog may prompt you to revisit the priorities of the backlog items.

The architecture backlog should be made available to anyone who may need to be aware of the design's progress. If you do have separate backlogs for the architecture and the rest of the project, keep in mind that the audience for the two backlogs may be different. It really depends on the project and the level of involvement and transparency that exists between the project team and other stakeholders involved with the project.

In some cases, clients may have access to the product backlog to track functionality, but the team may want to keep the architecture backlog private.

Summary

Software architecture design plays a critical part in the creation and success of software architectures. At its core, architecture design involves making design decisions to produce solutions to design problems. The result is an architecture design that can be validated, formally documented, and eventually used by development teams.

There are two main approaches to architecture design, the top-down and bottom-up approaches. We examined situations in which one would be used over the other and learned how a combination of the two approaches often works best.

Architectural drivers, which are the inputs into the architecture design process, guide the architecture design. They include design objectives, primary functional requirements, quality attribute scenarios, constraints, and architectural concerns.

Designing a software architecture can be challenging, but we can leverage design concepts, such as software architecture patterns, reference architectures, tactics, and externally developed software, to assist with the design of solutions.

While formal documentation of an architecture does not need to occur during the design process, documenting, such as sketching the design and recording the design rationale, should take place.

Following an architecture design process helps to guide software architects with their design. There are a number of architecture design processes that are available to use, so you'll have to do some research in order to select a process that will work best for your project. Architecture design processes can be modified and supplemented with other techniques and processes to fill in any gaps with the process that you want to use.

A way of prioritizing and tracking the progress of architecture work should be put into place, such as having a backlog specific to architecture.

In the next chapter, we will explore some of the principles and best practices of software development. Some of them can be applied to software architecture, while others may be concepts that you will want to communicate to your team and encourage them to use in their implementations.