12
Documenting and Reviewing Software Architectures

An important aspect of being a successful software architect is the ability to record and communicate your architecture to others. We will begin by exploring the reasons why we document a software architecture. You will then become familiar with **architecture descriptions** (**ADs**) and the architecture views that are a part of them.

The chapter will provide an overview of the **Unified Modeling Language** (**UML**), which is one of the more popular and widely used modeling languages. You will learn about some of the most common UML diagram types.

As parts of a software architecture design are completed, the development team and relevant stakeholders need to review the architecture to determine whether it will satisfy the functional requirements and quality attribute scenarios. We will detail several different architecture review methods.

In this chapter, we will cover the following topics:

- Uses of software architecture documentation
- Creating architecture descriptions (ADs), including architecture views
- Overview of the UML
- Reviewing software architectures

Uses of software architecture documentation

Some software projects skip software architecture documentation altogether or do it merely as an afterthought. Some of the projects that complete architecture documentation only do so because it is required. For example, it may be required by an organization's process or there may be a contractual obligation to provide it to a client.

However, good software architects understand the value of documenting their software architecture. Good documentation serves to communicate the architecture to others, assist the development team, educate team members, facilitate architecture reviews, allow for the reuse of architectural knowledge, and help the software architect.

Communicating your architecture to others

A good software architecture is only useful if it can be communicated to others. If it cannot be communicated to others effectively, then those who need to build, use, and modify the architecture may not be able to do so, at least not in the way that the architect intended. During software architecture design, we identify all of the structures that make up the architecture and how they interact with each other. Documentation allows us to communicate those structures and interactions. The artifacts that are created when an architecture is documented focus on communicating the solution (architecture) to various audiences, which can be made up of both technical and non-technical people.

Software architects will need to communicate their architecture to the development team, management, and other stakeholders. Different stakeholders will have different reasons and priorities behind wanting to learn about the architecture, but a software architecture is abstract enough that a variety of people can use it to reason about a software system. Different types of architecture view can help to communicate the architecture to everyone who needs to understand it.

Assisting the development team

Documentation is useful to the team during the design and development of the software system. Learning about the structures and elements of the architecture and how they interact with each other will allow developers to understand how they should be implementing their functionality. By seeing the interfaces available, developers will gain an understanding as to what needs to be implemented and what is available to use in completed implementations. The documentation enables developers to complete their work in a way that abides by the design decisions that have been made for the architecture.

Software architecture also restricts some of the design choices available to developers and puts constraints on implementation, reducing the complexity of the software system. Architecture documentation communicates design decisions, which helps to prevent developers from making the wrong decisions about how a piece of functionality should be implemented.

Educates team members

Software architecture documentation is also beneficial in educating the developers on the team. Developers who are unfamiliar with the system, either because it is a new one that has just begun its design process or because they have joined an existing project, can use it as a guide to become familiar with the architecture. It is useful for software developers to understand the design decisions that shape the architecture they are using.

As a software architecture changes, the documentation should be updated as well. Good documentation will help to communicate any changes to the development team so that they can be aware of them.

Providing input for software architecture reviews

Software architectures are reviewed to ensure that they have the capability to meet requirements, including quality attribute scenarios. Architecture documentation is useful during this process because it contains details that will allow a review team to analyze an architecture and make these types of determination.

The documentation can also be used to evaluate and compare alternative software architectures. Architecture documentation helps those who are tasked with comparing architectures to accomplish their work by providing the necessary details to make informed and accurate evaluations.

Allowing for the reuse of architectural knowledge

Software architecture documentation allows architectural knowledge to be reused for other projects. The design decisions made, the design rationale that formed the decisions, and any lessons learned can be leveraged when other software systems need to be created or maintained.

Reuse allows organizations to be more efficient and productive with their software development. If an organization is developing a software product line, which consists of multiple products from the same company to address a particular market, the software products may have some similar functional and non-functional requirements and may share a similar look and feel in terms of the user interface. Parts of an architecture made for one software product may be useful for one or more of the other ones. Software architecture documentation can facilitate the reuse of an architecture.

Help the software architect

Documenting a software architecture helps the software architect. Software architects will be asked plenty of questions about the architecture by a variety of stakeholders and the documentation can help to answer them. It supports software architects to fulfill some of their responsibilities by providing artifacts with which to communicate the architecture to others, assist the development team, educate team members, review the software architecture, and pass on architectural knowledge for reuse.

Some projects are rather complex and it can be difficult to remember all of the structures that make up the architecture and how they interact with each other, even for the software architect who was directly involved. The documentation can help to remind a software architect of this information if it has to be revisited months or even years later.

A software architect may be trying to get a project off the ground or to gain funding for their project. Solid documentation can help the software architect achieve these goals by providing information to stakeholders. Software architects can use some of the completed documentation when they have presentations to give about the software.

If a software architect leaves the project or the organization, the documentation that is left behind can answer questions when the software architect is no longer available.

Creating architecture descriptions (ADs)

An **architecture description** (**AD**) is a work product used to express and communicate an architecture. The actual architecture of a software system is separate from the artifacts that describe and document it, such that we can create different artifacts for a given architecture.

ADs identify the stakeholders of the software system and their concerns. Stakeholders include people such as users, administrators, domain experts, business analysts, product owners, management, and the development team.

Each of these stakeholders has various concerns, which are either unique to them or shared with other stakeholders. Examples of system concerns related to architecture include the goals of the system, the suitability of the architecture to accomplish those goals, the ability of the architecture to meet quality attribute scenarios, the ease with which the software system can be developed and maintained with the architecture, and any risks to the stakeholders. ADs consist of one or more architecture views, with the views addressing the different concerns of the stakeholders.

Software architecture views

Many software systems are complex, making them difficult to understand. This is particularly true when one attempts to look at the whole system at once. **Architecture views** are used to ease understanding, as each one focuses on a specific structure or structures of the architecture. This allows a software architect to document and communicate only a small piece of the architecture at a time. Having multiple views represent an architecture allows the software architect to communicate it in a manageable way.

Deciding which views to create depends on the goals of the documentation and the audience. In the *Uses of software architecture documentation* section earlier in this chapter, we covered some of the reasons that documentation is useful. Different views will focus on different aspects of the architecture, so the intended usage of the architecture documentation will dictate the types of view that you create.

There is no definitive list of views that must be created. The same goes for deciding how many views to create. Each software project is different, so there is no set number of views that are required for all of them. The number of artifacts that need to be created for a software architecture is the amount necessary to effectively communicate your architecture to the different audiences that are interested.

There is a cost associated with creating and maintaining a view, so we only want to introduce a view if it is needed and will provide a benefit. We never want the documentation to be insufficient but we also do not want to spend inordinate amounts of time creating too much of it.

During software architecture design, informal documentation, in the form of sketches, should be made to record design decisions. These sketches may include items such as the structures, elements, relationships between elements, and architecture patterns used. These sketches are important to document the work that was done during design. While they are not complete enough to be released as the final documentation, they can be used as the basis for the architecture views once it comes time to formally document the architecture.

Software architecture notations

There are different types of notation that can be used with software architecture views. They help software architects to communicate their design and some of them can be used by tools for code generation.

Notations differ from each other predominately based on their level of formality. Notations that are more formal typically require more effort, not just in creating the artifacts but also in understanding them. However, they can provide more detail and reduce ambiguities that may exist with less formal notations.

There are many different notations. Each one is good for something but none of them are good for everything. When deciding which notation to use for all or some of your views, consider the purpose of the diagrams, the stakeholders that will be examining them, your familiarity with the notation, and the tools that are available to create them. The three main types of software architecture notation are:

- Informal
- Semiformal
- Formal

Informal software architecture notations

Informal notations are views of the architecture that are often created with general-purpose tools, using whatever conventions the team deems appropriate. Using this type of notation helps communicate the software architecture to customers and other stakeholders who may not be as technical or have the need for more formalized approaches. Software architects may consider using this notation for artifacts created for the project's management team. They can help project management to understand the scope of the project.

Natural language is used with the notation, so these types of artifacts cannot be formally analyzed and tools will not be able to automatically generate code from the artifacts.

Semiformal software architecture notations

A semiformal notation in views is a standardized notation that can be used in diagrams. However, fully-defined semantics are not part of a semiformal notation. Unlike informal notations, this type of notation allows for some level of analysis and can potentially be used to automatically generate code from the models.

UML is an example of a semiformal notation that is very popular for modeling. There are some notations that can be used with UML that extend it to provide more robust semantics, making UML more formalized.

Formal software architecture notations

Views of the architecture that use formal notations have precise semantics, usually mathematically based. This allows for formal analysis of both the syntax and the semantics. Tools can use artifacts created with a formal notation for automated code generation.

Some software projects choose to not use a formal notation because they require more effort to use, they require certain skills on the part of those creating the architecture views, and stakeholders may find them difficult to understand. Formal notations are not as useful for communication with non-technical stakeholders.

A number of formal notations exist, including various **architecture description languages** (**ADLs**). An ADL is a formal type of expression that is used to represent and model a software architecture. **Architecture Analysis and Design Language** (**AADL**) and **Systems Modeling Language** (**SysML**) are two examples of ADLs. AADL was originally created to model both hardware and software. It can be used to describe a software architecture, create documentation, and generate code.

The book *Documenting Software Architectures – Views and Beyond* describes AADL as follows:

> *"The AADL standard defines a textual and graphical language to represent the runtime architecture of software systems as a component-based model in terms of tasks and their interactions, the hardware platform the system executes on, possibly in a distributed fashion, and the physical environment it interfaces with, such as a plane, car, medical device, robot, satellite, or collections of such systems. This core language includes properties concerning timing, resource consumption in terms of processors, memory, network, deployment alternatives of software on different hardware platforms, and traceability to the application source code."*

SysML is a general-purpose, graphical modeling language for systems. It is a standard that is maintained by the Object Management Group (OMG). It can be used for a number of activities, including specification, analysis, design, and verification.

SysML is a subset of UML and reuses some of the same diagram types. We will discuss UML in more detail shortly but here is a list of the SysML diagram types that are reused from UML without any modification:

- Use case diagrams
- Sequence diagrams
- State diagrams
- Package diagrams

The following diagrams have been modified in SysML from their UML counterparts:

- Activity diagrams
- Block definition diagrams
- Internal block diagrams

SysML also introduces some new diagram types that do not exist in UML:

- Requirement diagrams
- Parametric diagrams

Including design rationales

An architecture description should include the **design rationale** behind the design decisions being documented. In `Chapter 5`, *Designing Software Architectures*, we discussed the design rationale, which is an explanation that contains the reasons and justification for design decisions related to the architecture. Without documenting the design rationale, the reasons that a design decision was made will not be known. Recording the design rationale is beneficial even for those who are involved in a design decision, as the details of a decision can be forgotten over time.

It is not necessary (or practical) to record every design decision that is made but any decisions that are important to the architecture are candidates to be documented. When documenting a design rationale, keep in mind that in addition to including design decisions, it is sometimes useful to include details on why alternative approaches were not taken and why certain design decisions were not made at all.

Overview of the Unified Modeling Language (UML)

The **Unified Modeling Language** (**UML**) is a general-purpose, standardized modeling language. It is widely used and understood, making it a popular choice for modeling a software architecture. While this section is not intended as an exhaustive tutorial on UML, we will cover some of the most popular UML diagrams and their purpose. If you are already familiar with UML or prefer to use a different modeling language, feel free to skip this section.

Types of modeling

In UML, there are two main types of modeling: *structural modeling* and *behavioral modeling*. Structural modeling focuses on the static structure of the system, its parts, and how they are related to each other. They do not show details about the dynamic behavior of a system. Some of the structure diagrams in UML include:

- Class diagrams
- Component diagrams
- Package diagrams
- Deployment diagrams

Behavioral modeling shows the dynamic behavior of the components in a system. Unlike the static nature of structure diagrams, behavior diagrams describe changes to the system over time. Some of the behavior diagrams in UML include:

- Use case diagrams
- Sequence diagrams
- Activity diagrams

Class diagrams

Classes are templates (blueprints) for creating objects in a software system. They include attributes (member variables) that hold the state of an object and operations (methods) that represent behavior. **Class diagrams**, which are among the most popular of UML diagrams, show us the structure of a software system by allowing us to see the classes and their relationships. A number of team members may find class diagrams useful, including the software architect, developers, QA personnel, operations engineers, product owners, and business analysts.

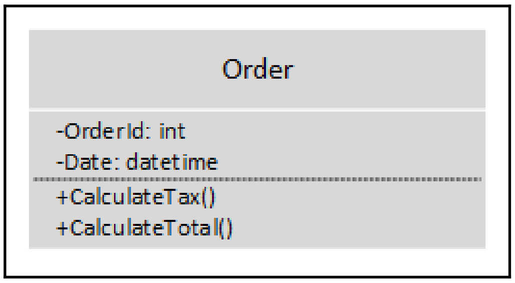
A rectangle is used in a class diagram to graphically represent a class and each one can have up to three sections in it. The upper section shows the name of the class, the middle section contains the attributes of the class, and the bottom section details the operations.

Visibility

Visibility dictates the accessibility of a member (attribute or operation) and can be designated by placing a notation before the member's name. In general, you want to give only as much accessibility as is needed. The following table details the most common visibility notations:

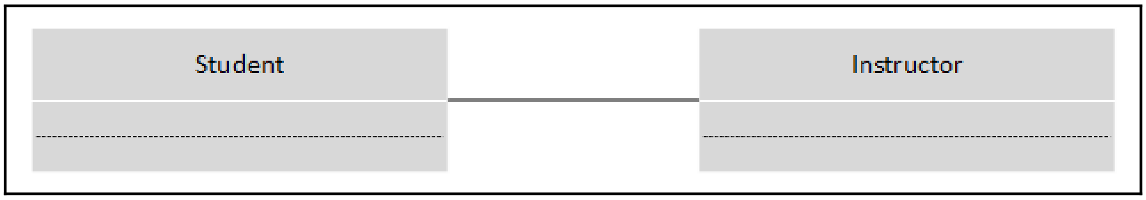
Notation	Visibility	Description
+	Public	Member is accessible by other types.
#	Protected	Member is accessible within the same type as well as types that inherit from it.
~	Package	Member is accessible from any type within the same package. It is not accessible from outside the package, even if it is an inheriting class.
-	Private	Member is accessible only within the type that declares it.

For example, the following diagram shows the **Order** class, which has two private attributes, **OrderId** and **OrderDate**, as well as two public operations, **CalculateTax** and **CalculateTotal**:



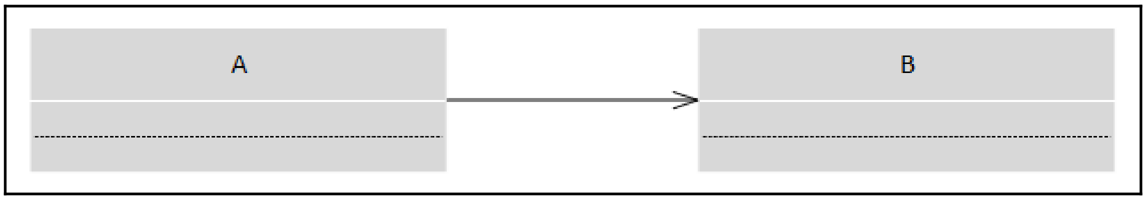
Association

An association is a broad term that refers to a semantic relationship between classes. If one class uses another class (unidirectional), or two classes use each other (bidirectional), they have a relationship. Relationships between classes are represented by an association, which is shown on a class diagram as a solid line:



When there are no arrowheads at the end of the line, the navigability of the association is unspecified. However, in diagrams where arrows are used for one-way navigable associations, a line with no arrows is assumed to represent bidirectional navigability. In the preceding example, which would mean that both **Student** and **Instructor** are accessible from each other. Alternatively, a bidirectional association can be depicted by having an open arrowhead at the end of both lines.

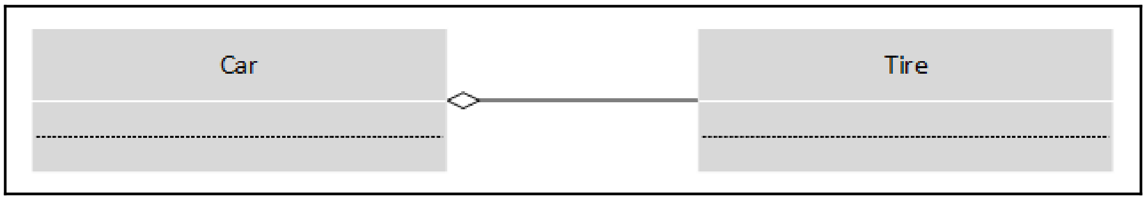
If we want to model unidirectional navigability, an open arrowhead can be used. In the following diagram, class **B** is navigable from class **A**:



Aggregation and composition are subsets of association and are used to represent specific types of association.

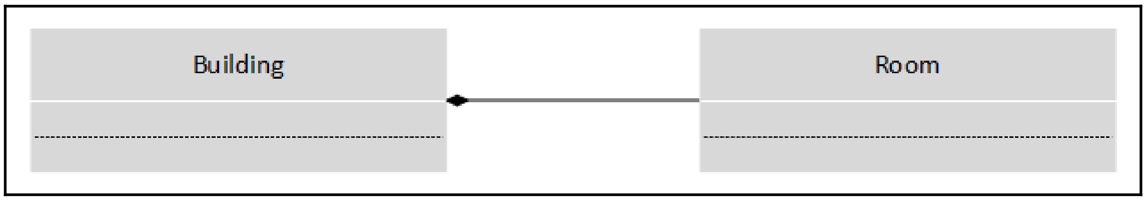
Aggregation

Aggregation is a relationship in which a child object can exist independently of the parent. It is graphically represented by a hollow diamond shape. For example, in a domain with a **Tire** object, we can say that, even though a **Car** object has Tire objects, a **Tire** object can exist without a **Car** object:



Composition

Composition is a relationship in which an object cannot exist independently of another object. It is graphically represented by a filled diamond shape. For example, in a domain with a **Room** object, we might say that a **Room** object cannot exist without a **Building** object:

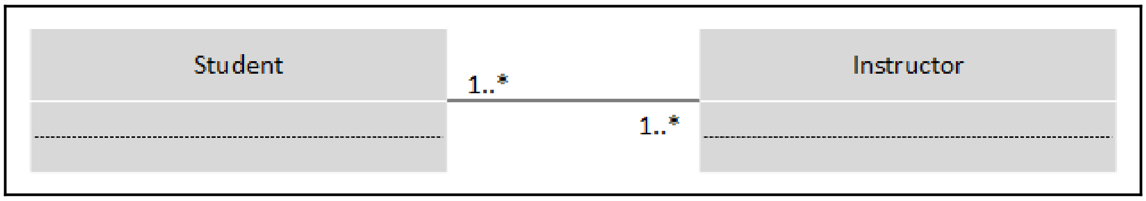


Multiplicity

Multiplicity allows you to define the cardinality of a relationship between classes. The multiplicity of a relationship describes the number of objects that can participate in it. The following table shows the different types of multiplicity that can be specified:

Notation	Multiplicity
0..1	Zero or one
1	One and only one
1..1	One and only one
0..*	Zero or more
*	Zero or more
1..*	One or more

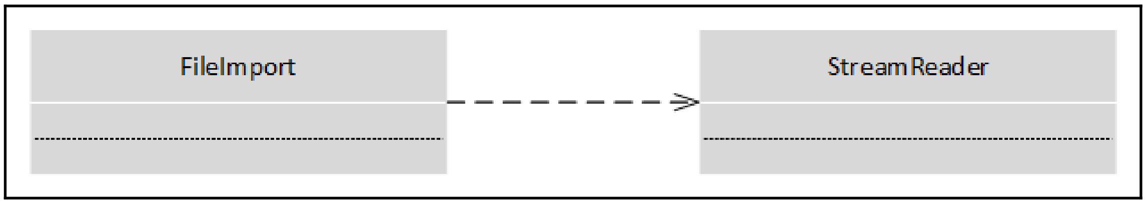
For example, the following diagram depicts that each **Student** is taught by one or more instructors and that each **Instructor** teaches one or more students:



Dependency

A dependency is a type of relationship between UML elements, such as classes, in which one element requires, needs, or depends on another element. The dependency is sometimes referred to as a supplier/client relationship because the supplier provides something to the client. The client is either semantically or structurally dependent on the supplier. A dependency may mean that changes to a supplier may require changes to a client.

In an association, one class may have a reference to the other as a member variable. A dependency relationship is slightly weaker. For example, a dependency may exist because a return type or parameter for a method in one class references another class:



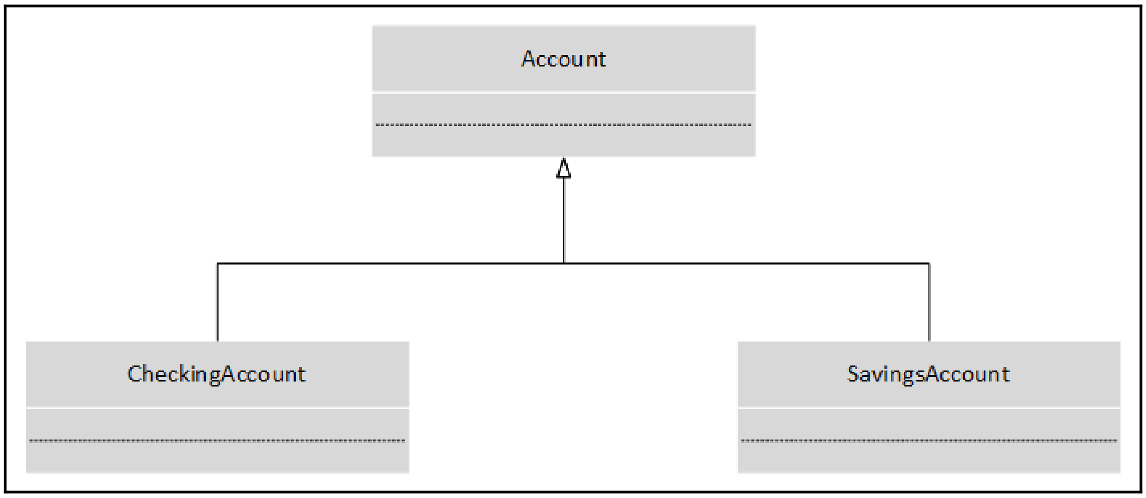
A dependency is graphically represented by a dashed line with an open arrowhead. In the preceding example, the **FileImport** class (client) depends on the **StreamReader** class (supplier). There can be many dependencies on a single diagram, so you may not want to show every dependency. However, you should show that are important to what you are trying to communicate in a particular diagram.

Generalization/specialization

Generalization is the process of abstracting common attributes and operations into a base class. The base class is sometimes referred to as the superclass, base type, or parent class. Generalization is also known as inheritance. The base class contains general attributes, operations, and associations that are shared with all of its subclasses. Generalization is graphically represented with a hollow triangle on the part of the connecting line that is closest to the base class.

Specialization is the converse of generalization in that it involves creating subclasses from an existing class. Subclasses are sometimes referred to as a derived class, derived type, inheriting class, inheriting type, or child class.

For example, our domain may have different types of account, such as a checking account and a savings account. These classes may share some of the same properties and behaviors. Rather than repeating what is shared in each of these account classes, our model may have an **Account** base class, which contains the generalized attributes and operations that are common to all account classes:

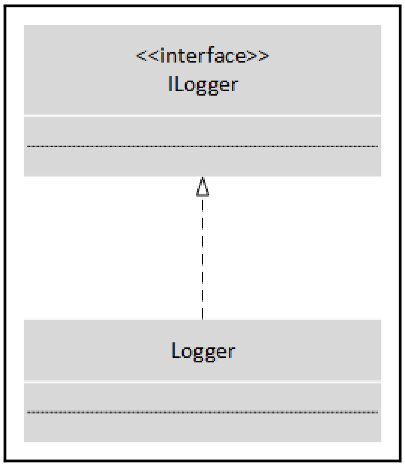


The **CheckingAccount** and **SavingsAccount** classes, which are subclasses, inherit from the **Account** class and demonstrate an *is a* relationship. **CheckingAccount** *is an* **Account**, just as **SavingsAccount** *is an* **Account**. **CheckingAccount** and **SavingsAccount** are specializations of **Account**.

Depending on the programming language used for implementation, it may be possible to allow some attributes or operations to be overridden in subclasses. Subclasses can also introduce their own specialized attributes, operations, and associations that are specific to their class.

Realization

Realization denotes a relationship in which one element *realizes* or implements the behavior that another element specifies. A common example of this is when a class implements an interface. Realization is graphically represented with a hollow triangle at the end of a dashed line, with the hollow triangle appearing closest to the element that is specifying the behavior.



In the preceding diagram, you may have noticed that **ILogger** is designated as an interface. This designation is done through a *stereotype*. Stereotypes are one of the extensibility mechanisms available in UML, which allow you to extend vocabulary and introduce new elements. In this case, a stereotype has been used to indicate that **ILogger** is an interface.

Stereotypes are graphically represented by enclosing the name in guillemets (angle quotes). They are similar to the symbols for less than and greater than, which can be used if guillemets are unavailable.

Component diagrams

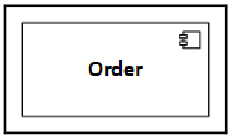
Component diagrams detail the structural relationship between components of a system. These diagrams are typically needed with complex software systems that consist of many components, as it is helpful to view the components and their relationships. They essentially depict how the components of a software system are wired together, which is why they are sometimes referred to as *wiring diagrams*.

Component diagrams help us to identify the interfaces between different components of our software system. Components communicate with each other through interfaces, and component diagrams allow us to see system behavior as it relates to an interface. Interfaces define a contract by defining the methods and properties that are required for implementations. Implementations can be changed as long as the classes that are dependent on them are coded for interfaces and not for specific implementations.

By identifying interfaces, we are able to identify the replaceable parts of our software system. Having this knowledge gives us the ability to know where we can potentially reuse a component that the organization has already created or where a third-party component could be used. Components that we create for a software system may also be leveraged in other software applications that the organization has in development or will develop in the future.

Knowing the components of a software system also makes it easier for project decision makers to divide up the work. Once the interface is agreed upon, one or more developers on the team, or even a separate team, can work independently of others in developing a component.

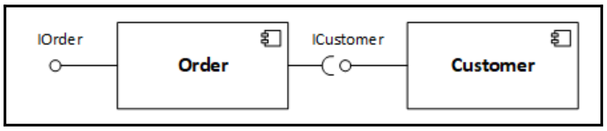
In a component diagram, components are graphically represented as a rectangle with a *component* symbol (a rectangular block with two smaller rectangles on the left side). For example, an **Order** component would look like the following:



Alternatively, the component stereotype can be used, either in addition to the component symbol or in place of it, to designate that an object on the diagram is a component.

Interfaces that a component provides are graphically represented by a *small circle* at the end of a line, which is also sometimes referred to as the *lollipop* symbol. Interfaces that a component requires are represented by a *half circle* at the end of a line, which is also referred to as a *socket*.

For example, let's say that our **Order** component implements the **IOrder** interface and requires an implementation of **ICustomer**. The **Customer** component implements the **ICustomer** interface:



Keep in mind that components can contain other components. For example, we could model an **Order** system component that contains, within it, the **Order**, **Customer**, and other components, along with all of their relationships.

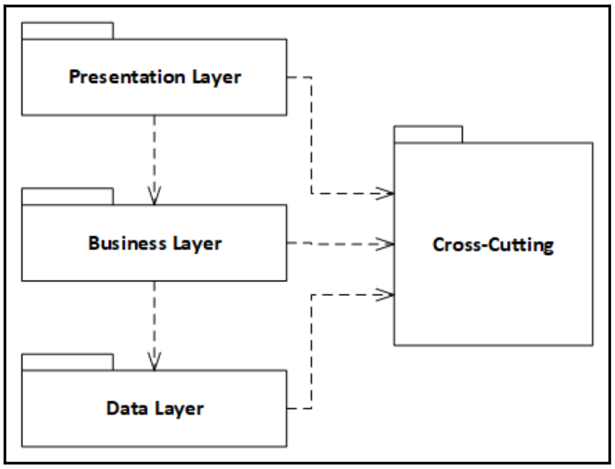
Package diagrams

In UML, packages logically group elements together and provide a namespace for the groups. Many UML elements, such as classes, use cases, and components, can be grouped together in packages. Packages can also contain other packages. **Package diagrams** are used to show the dependencies between packages in a software system.

The more complex a software system is, the more difficult it can be to understand all of the models. Package diagrams make it easier for people to reason about large, complex systems by grouping elements together and allowing us to see the dependencies.

In addition to modeling standard dependencies, we can model *package import* and *package merge* types of dependencies. A *package import* is a relationship between an importing namespace and an imported package. This can allow us to reference package members from other namespaces without fully qualifying them. A *package merge* is a relationship between two packages in which one package is extended by the contents of another package. The two packages are essentially combined.

A package is graphically represented in UML by a symbol that looks like a file folder with a name. The following is an example of a high-level package diagram for a layered application:



By looking at it, we can see that the **Presentation Layer** depends on the **Business Layer**, the **Business Layer** depends on the **Data Layer**, and all three layers depend on the **Cross-Cutting** package.

Deployment diagrams

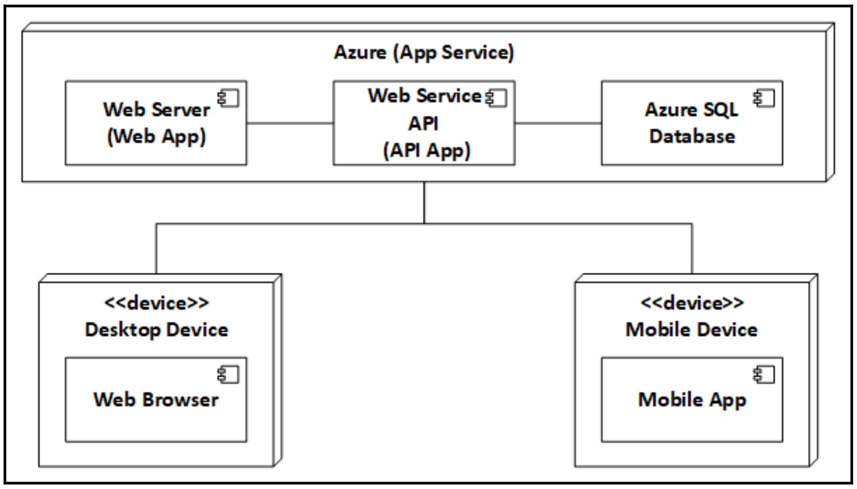
Deployment diagrams represent the physical deployment of artifacts on nodes. An a*rtifact* is a physical piece of information, such as a source code file, binary file, script, table in a database, or document.

A *node* is a computational resource that artifacts are deployed on for execution. Nodes can contain other nodes. There are two types of node: device nodes and **execution environment nodes** (**EENs**).

- **Device**: A device represents a physical computational resource (hardware) that can execute a program. Examples include a server, laptop, tablet, or mobile phone. Devices may consist of other devices.
- **Execution environment**: An execution environment is a software container that resides in a device. It provides an execution environment for artifacts that are deployed on it. Examples include an operating system, a JVM, or a Docker container. Execution environments can be nested.

Deployment diagrams are used to show the software elements in an architecture and how they will be deployed to hardware elements. They provide a view of the hardware and the system's topology.

In a deployment diagram, nodes are graphically represented by a three-dimensional box. In the following example, there are three nodes. One represents the **Azure (App Service)**, one is **Desktop Device**, and one is **Mobile Device**. Notice that a **<<device>>** stereotype is used to indicate that a node is a device. The various nodes in the example contain components and the lines show associations between the nodes:

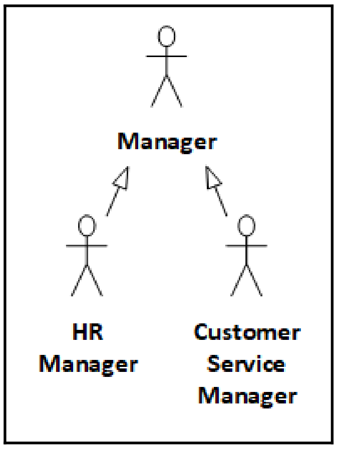


Use case diagrams

Use cases are text that describes a software system's behavior as it responds to requests from system users, known as *actors*. An actor is the role for someone or something that interacts with the system and can be a person, organization, or an external system.

Just like with classes in a class diagram, generalizations can be done on actors. *Actor generalization* is a relationship between actors in which one actor (descendant) inherits the role and properties from another actor (ancestor).

For example, if our domain had different types of managers, such as an **HR Manager** and a **Customer Service Manager**, they may both inherit from a **Manager** ancestor actor:

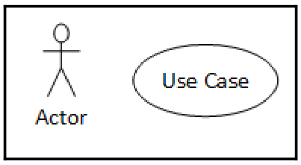


Actor generalization is graphically represented in the same way that generalization is with classes. It is done with a hollow triangle on the part of the connecting line that is closest to the *ancestor* actor.

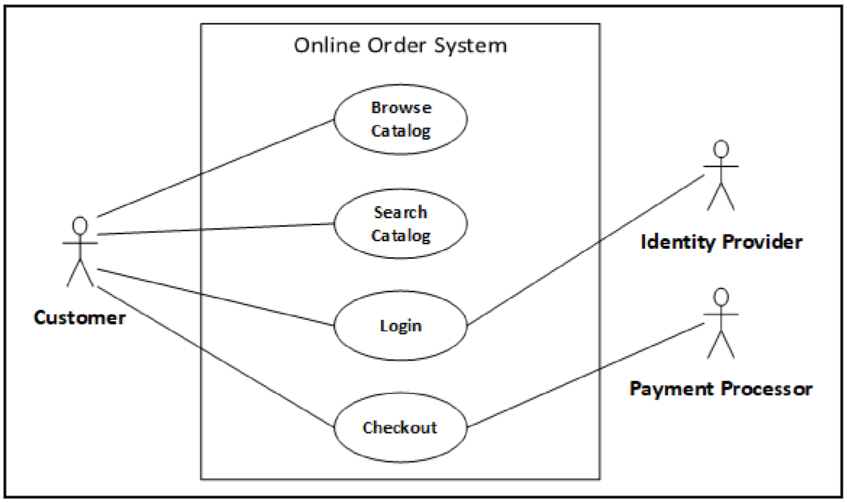
Use cases are something that the system does or something that happens to the system. Use cases should be easy to read and are usually brief. Actors have goals and use cases describe ways to carry out those goals by using the software system.

A **use case diagram** is a graphic representation of a use case, the relevant actors, and their relationships. It details the actors and how they interact with the software system. Use case diagrams allow people to understand the scope of the software system and the functionality that will be provided to actors. They can be useful for traceability in that we can verify that a software system is meeting its functional requirements.

In a use case diagram, actors are typically represented by a stick figure with the name of the actor's role appearing underneath it. Use cases are graphically represented with a horizontally shaped oval. The name of the use case appears inside the oval:



Lines are used to show associations between actors and use cases. Use case diagrams can describe context by showing the system's scope. A system boundary box can be used to present what is part of the system and what is external to it:



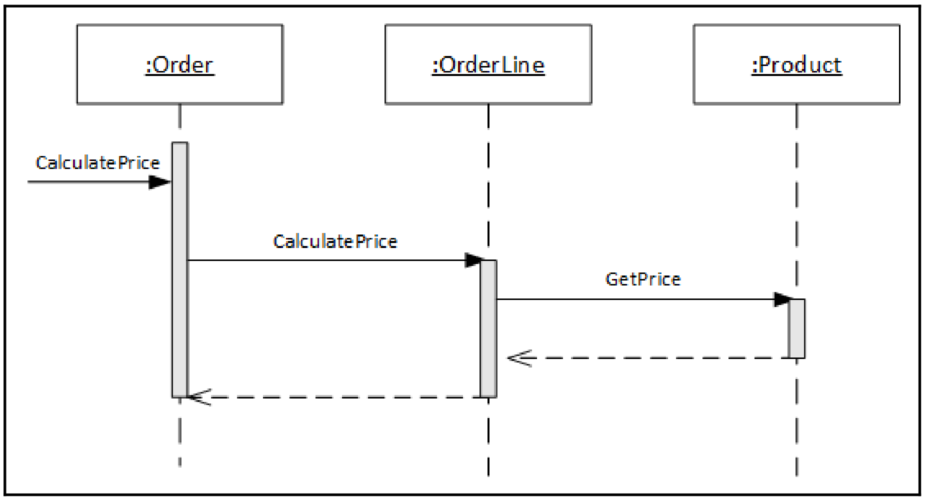
In the preceding diagram, there are three actors, all of which are external to the **Online Order System**. The **Customer** actor is a person but the **Identity Provider** and **Payment Processor** actors are external systems.

There are four use cases shown in this simplified example. The **Customer** is associated with all of them but the **Identity Provider** is only involved with the **Login** use case and the **Payment Processor** is only associated with the **Checkout** use case.

Sequence diagrams

Sequence diagrams model how components in a software system interact and communicate. They are one type of interaction diagram, which is a subset of behavior diagrams. Other interaction diagrams include the communication diagram, timing diagram, and interaction overview diagram.

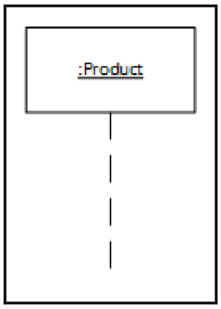
Sequence diagrams describe a sequence of events from the software system. The following example shows the flow of logic within a system for price calculation:



Sequence diagrams are sometimes referred to as event diagrams or event scenarios. They can be used to see how components interact with each other and in what order they do so. Some examples of what you might want to model using a sequence diagram include usage scenarios, service logic, and method logic.

Lifeline

In sequence diagrams, an object is graphically represented by a rectangle with its *lifeline* descending from the center of its bottom:



The lifeline shows the lifespan of the object and is represented by a vertical dashed line. The passage of time starts at the top of the line and goes downward. The rectangle can show both the name of the object and the class, separated by a colon, as follows:

```
objectname : classname
```

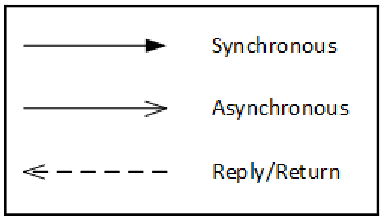
The object and class names are underlined. A lifeline can represent an object or a class. An object can be left unnamed if we are modeling a class or in cases where the object's name is unimportant. In that situation, you will simply see a colon followed by the class's name. Some people prefer to just see the class's name with no colon. If you are modeling objects and want to differentiate between different objects of the same class, you should specify an object name. When diagramming objects/classes, the attributes and operations of an object are not listed.

Activation boxes

Activation boxes on the lifelines show when an object is completing a task. For example, when a message is sent to an object, the time period from when the message is received until the response is returned can be represented with an activation box. Since activation boxes are on the lifelines, they also represent time. The longer the activation box, the longer the task will take to complete.

Messages

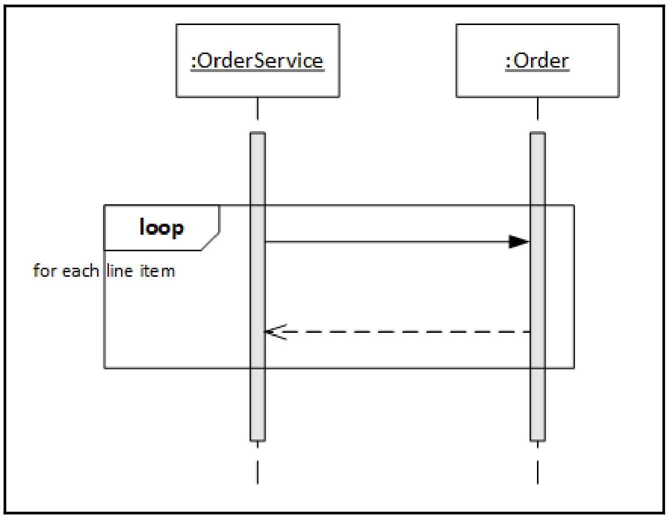
Arrows are used to graphically represent messages that are passed between objects. The type of arrow indicates the type of message that is being passed:



Synchronous messages are shown as a solid line with a solid arrowhead. A synchronous message is one in which the sender must wait for a response before it can continue. **Asynchronous** messages are represented by a solid line with a lined arrowhead. With an asynchronous message, the sender does not have to wait for a response before continuing. **Reply/Return** messages and asynchronous return messages are both represented by a dashed line with a lined arrowhead.

Loops

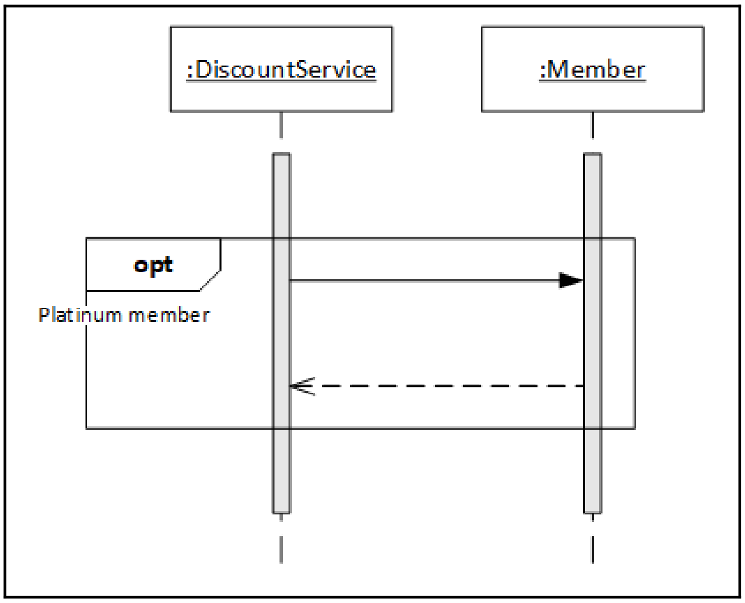
In order to model a loop in a sequence diagram, a box is placed over the part of the diagram that is iterating through a loop. An inverted tab at the top-left corner of the box is labeled with the word **loop** to signify that the structured control flow is a loop. The subject that is being iterated over is commonly labeled with a guard message that is placed below the inverted tab. In the following example, the logic is iterating over each line item in an order:



Optional flows

In a sequence diagram, you may need to model optional flows. These represent logic that will optionally be executed based on some condition. Similar to loops, an optional control flow is graphically represented with a box that is placed over the part of the diagram that is related to the optional flow. An inverted tab at the top-left corner of the box is labeled with **opt** to denote that it is an optional flow.

The condition for the optional flow can be labeled by using a guard message that is placed below the inverted tab. In the following diagram, an optional flow is executed only if a member is a platinum member:

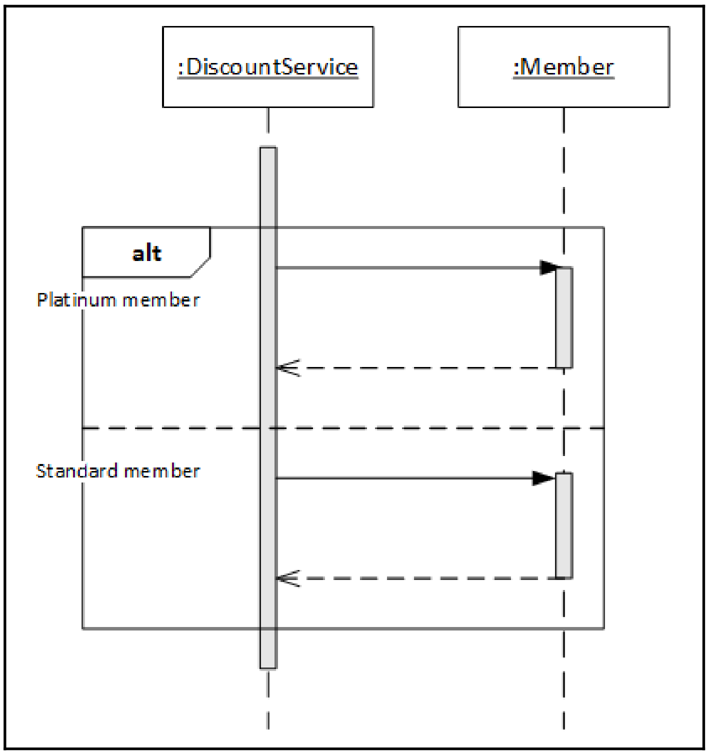


Alternative flows

When you want to model alternative (conditional) fragments in a sequence diagram, a box can be placed over the part of the diagram that captures the alternatives. An inverted tab at the top-left corner of the box is labeled with **alt** to denote that it is an alternative fragment.

Alternative flows are similar to optional ones so the two should not be confused with each other. While an optional flow checks a single condition and may or may not execute a fragment, alternative flows offer multiple possibilities. Only the alternative fragment whose condition is true will execute.

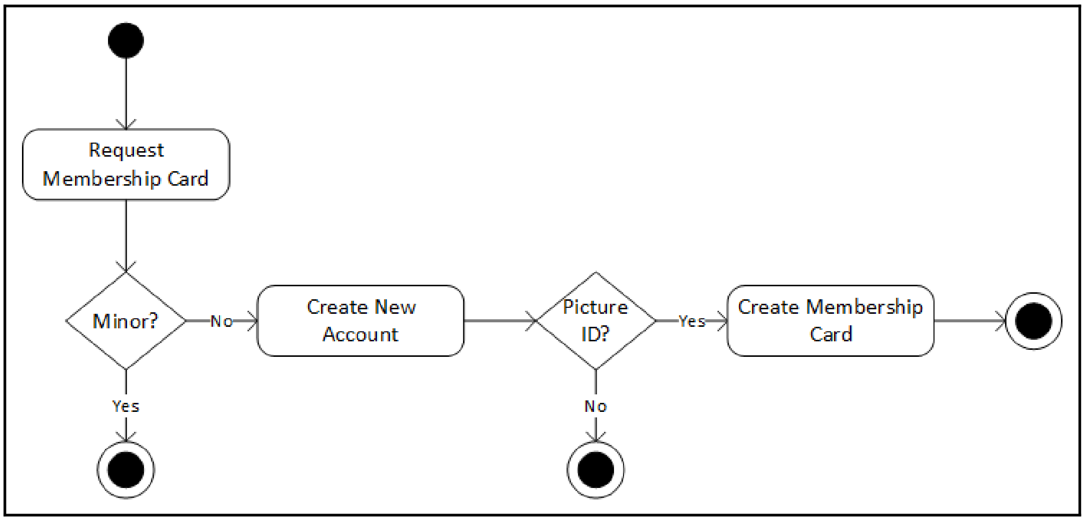
A guard message can be placed at the start of each alternative to describe the condition and a dotted line is used to separate each alternative. In the following diagram, there are two alternatives, one for platinum members and one for standard members:



Activity diagrams

An **activity diagram** allows us to visually represent a series of actions in the form of a workflow. It shows a control flow and is similar to a flowchart. Activity diagrams can be used to model things such as a business process, flow within a use case, and procedural logic.

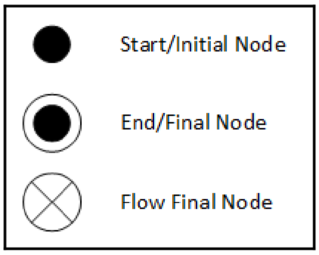
The following diagram shows the workflow for creating a new membership card:



The activities in an activity diagram can either be sequential or concurrent. An activity is shown as a rectangle with rounded corners. The rectangle encloses all of the elements of an activity, such as its actions and control flows.

Start/end nodes

Some of the nodes that can appear in an activity diagram represent the different ways in which flows can begin and end:

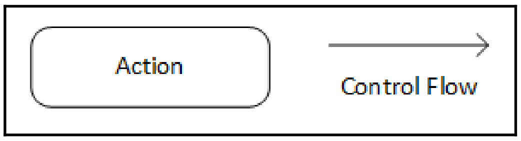


An activity diagram begins with an initial state, or start point, which is graphically represented by a small, solid circle (**Start/Initial Node**). The activity diagram ends with a final state that is graphically represented by a small, filled circle inside another circle (**End/Final Node**).

A **Flow Final Node**, which is a circle with an *X* inside, can be used to represent the end of a specific process flow. Unlike the end node, which denotes the end of all control flows within an activity, a flow final node represents the end of a single control flow.

Actions/Control flow

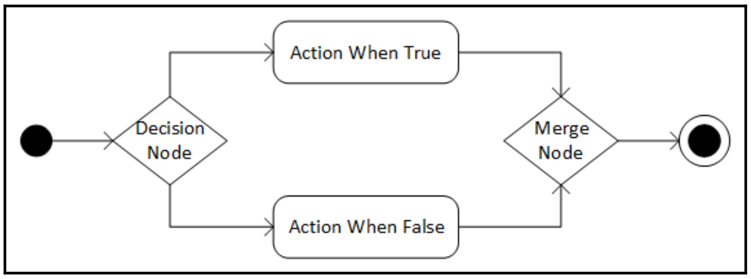
Actions are single steps within an activity. Like activities, they are also represented as a rectangle with rounded corners. A solid line with an open arrowhead is used to show control flow:



Decision/merge nodes

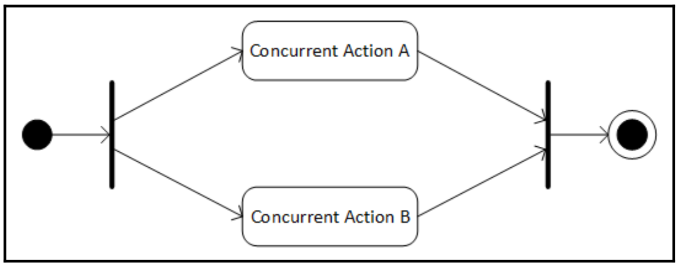
A decision occurs in a flow when there is some condition and there are at least two paths that branch from that decision. A label can be placed on each of the different branches to indicate the guard condition that would allow control to flow down the branch.

When you want to bring multiple alternate flows back to a single outgoing flow, a merge node is used. Both decision and merge nodes are graphically represented with a diamond symbol:



Fork/join nodes

When you want to model a single flow forking into two or more concurrent flows, you use a fork node. When you want to combine two or more concurrent flows back into a single outgoing flow, you use a join node. The following diagram illustrates a flow that has a fork and join node:



Both fork and join nodes are graphically represented with either a horizontal or vertical bar. The orientation of the bar is dependent on whether the flow is going from top to bottom or left to right. When fork and join nodes are used together, they are sometimes referred to as *synchronization.*

Reviewing software architectures

An important step in designing a high-quality software architecture is for it to go through a review process. Architecture reviews may also be conducted when an organization acquires software or to compare architectures. A review will determine whether the functional requirements and quality attribute scenarios can be satisfied with the software architecture. Reviewing the architecture helps the team find mistakes and correct them as early as possible. This can greatly reduce the amount of effort it takes to fix a defect and can help to avoid further rework.

In this section, we will be taking a look at the following software architecture evaluation methods:

- Software architecture analysis method (SAAM)
- Architecture tradeoff analysis method (ATAM)
- Active design review (ADR)
- Active reviews of intermediate designs (ARID)

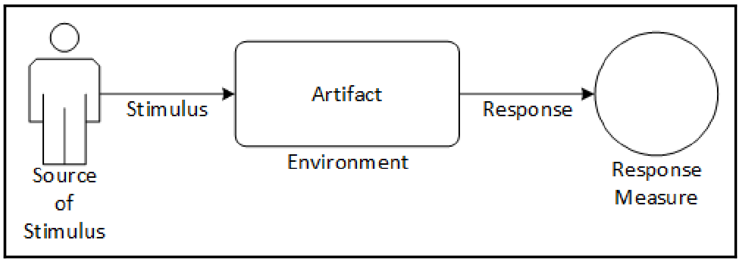
Software architecture analysis method (SAAM)

The **software architecture analysis method** (**SAAM**) is one of the first documented methods for evaluating software architectures. The original purpose of SAAM was to assess the modifiability of a software system, although some have extended it to review a software architecture for a variety of quality attributes, including reliability, portability, extensibility, and performance.

Scenario-based analysis of software architecture

SAAM is a scenario-based review method, and can be an effective way to review a software architecture. A scenario is a description of the interaction between some source, such as a stakeholder, and a software system. It represents some use or expected quality of the software system and may consist of a sequence of steps detailing the use or modification of it.

Scenarios can be used to test software quality attributes, which is one of the purposes behind software quality attribute scenarios:



A software quality attribute scenario consists of the following parts:

- **Source of stimulus**: The source is some entity, such as a stakeholder or another software system, which generates a particular stimulus.
- **Stimulus**: The stimulus is some condition that requires a response from the software system.
- **Artifact**: The artifact is the software that is stimulated. It can be a part of the software system, the entire software system, or a collection of multiple systems.
- **Environment**: The environment is the set of conditions under which the stimulus occurs. For example, a particular configuration of the software or specific values in the data may be necessary for the stimulus to exist.

- **Response**: The response is the activity that takes place when the stimulus arrives at the artifact.
- **Response Measure**: When a response occurs, it should be measurable. This allows us to test the response to ensure that the software system meets the requirements.

SAAM steps

There are six main steps in the software-architecture analysis method: develop scenarios, describe the architecture, classify and prioritize scenarios, evaluate scenarios, assess scenario interactions, and create an overall evaluation.

Step 1 – Develop scenarios

Using requirements and quality attributes, we can identify the different types of functionality that the software system is supposed to support and the qualities it is expected to have. This knowledge forms the basis for developing scenarios. Quality attribute scenarios, with the source of the stimulus, stimulus, artifact, environment, response, and response measure defined for each one, provide the type of information that makes scenarios useful when reviewing an architecture.

A variety of stakeholders should participate in brainstorming for scenarios, as the different perspectives and needs of a diverse group of people with an interest in the system will help to ensure that no important scenarios are overlooked. It is often useful to take an iterative approach when developing scenarios because identifying scenarios can lead the software architect, development team, and other stakeholders to think of additional scenarios.

Step 2 – Describe the architecture

In this step, the software architect describes the architecture to the review team. Completed architecture documentation can be used as part of the presentation. Any notations used in the documentation should be well understood by all of the review participants.

Step 3 – Classify and prioritize scenarios

Each scenario that is created in *Step 1 – Develop scenarios*, is classified and prioritized in this step. Scenarios can either be *direct* or *indirect* scenarios. If the software system does not require any modifications to perform the scenario, it can be classified as a direct scenario. If the scenario is not directly supported, meaning that some change has to be made to the software system for the scenario, then it is an indirect scenario.

Once scenarios have been classified, they should be prioritized based on importance. This can be accomplished by using some type of voting procedure. The scenarios that are determined to be of highest priority to the review team as a whole can be used for the evaluation.

Step 4 – Evaluate scenarios

In this SAAM step, the scenarios are evaluated. For each of the direct scenarios, the software architect demonstrates how the architecture can execute it. For any indirect scenarios, the team should identify what has to be changed (for example, modification/addition/deletion of components) in order to execute each one. The team should estimate the level of effort necessary to change the system so that it can execute indirect scenarios.

Step 5 – Assess scenario interaction

In this step, the reviewers analyze the interaction of the scenarios. If multiple *related* scenarios interact with the same component, this may be acceptable. However, if multiple *unrelated* scenarios interact with the same component, it could be an indication of a poor design. Further analysis should be conducted to determine whether the component is lacking a clear separation of responsibilities.

Refactoring may be necessary to avoid different scenarios interacting with the same component. The component may have low cohesion, indicating that its elements are not closely related. It may also exhibit tight coupling, signifying that the component is highly dependent on another component. Low cohesion and tight coupling increase complexity and reduce the maintainability of the system. If such a situation exists, a component may need to be separated into multiple components.

Step 6 – Create an overall evaluation

With the prior steps completed, the review team should have a list of scenarios that have been classified, prioritized, and evaluated. The interaction of the scenarios may reveal potential issues with the design. Ultimately, the review team must make a decision as to whether the architecture is viable and can be accepted as is or if it has to be modified in some way.

Architecture tradeoff analysis method (ATAM)

The **architecture tradeoff analysis method** (**ATAM**) is another scenario-based architecture review method. ATAM is a successor to SAAM and improves upon it. ATAM has a focus on reviewing design decisions and quality attributes.

ATAM participant roles

The main participant roles during ATAM evaluation are the evaluation team, project decision makers, and stakeholders. The evaluation team, which should ideally be a group that is external to the software project, consists of a team leader, evaluation leader, scenario scribe, proceedings scribe, and questioner.

- **Team leader**: The team leader coordinates and sets up the review. They are responsible for creating the evaluation team as well as ensuring that the final report is produced.
- **Evaluation leader**: The evaluation leader runs the actual review. This includes facilitating sessions that create/prioritize/select/evaluate scenarios.
- **Scenario scribe**: The scenario scribe writes notes about the scenarios on a whiteboard or flipchart as the evaluation is taking place.
- **Proceedings scribe**: The proceedings scribe is responsible for capturing notes in electronic format. Details about scenarios are an important aspect of the evaluation that the proceedings scribe will capture.
- **Questioner**: The questioner focuses on raising issues and asking questions related to the architecture, with a particular focus on quality attributes.
- **Project decision makers**: The project decision makers are the individuals who have the authority to make changes to the software if necessary, including the power to assign/approve resources for work. Project sponsors, project managers, and software architects typically make up the project decision-maker group.
- **Stakeholders**: The stakeholders include anyone who has a vested interest in the software architecture and the system as a whole.

ATAM phases

There are four main phases involved with an ATAM evaluation of a software architecture:

- **Phase 0**: Partnership and preparation
- **Phase 1**: Evaluation

- **Phase 2**: Evaluation (continued)
- **Phase 3**: Follow-up

Phase 0 – Partnership and preparation

This initial phase is used to prepare for the evaluation. The leader of the evaluation team meets with the project decision makers to agree on details about the evaluation. An agreement should be reached on the logistics of the meeting as well as which stakeholders will be invited.

As part of the preparation, the evaluation team looks at the architecture documentation to become familiar with the software application and its architecture. Expectations are set by the evaluation team as to the information they expect to be presented during *Phase 1*.

Phase 1 – Evaluation

Phase 1 is the first of two phases dedicated to the evaluation of the architecture. In this phase, the evaluation team meets with the project decision makers. *Phase 1* consists of the following steps:

1. Present the ATAM
2. Present the business drivers
3. Present the architecture
4. Identify architectural approaches
5. Generate the quality attribute utility tree
6. Analyze architectural approaches

Step 1 – Present the ATAM

In this step, the evaluation leader explains the ATAM to the project decision makers. Any questions about the ATAM can be answered during this step.

If everyone in the meeting is already familiar with the ATAM, this step could potentially be skipped. For example, a development team may go through the ATAM phases and steps multiple times as it iteratively designs the architecture. If the team consists of the same members, it may not be necessary to go over the ATAM each time. However, if any participants in a given iteration are new to the method, either this step should not be skipped or there must be a suitable alternative for participants who are new to the method to learn it.

Step 2 – Present the business drivers

This step is used to present the software system from a business perspective to the various participants. The business goals, functionality, architectural drivers, and any constraints will help everyone to understand the overall context of the software system. This information is presented by one of the project decision makers.

Step 3 – Present the architecture

The software architect presents the architecture to the participants in this step. The software architect should provide sufficient detail about the architecture so that the participants can understand it.

The level of detail needed in the presentation can vary from project to project. It really depends on the quality attribute scenarios of the system, how much of the architecture design is complete/documented, and how much time is available for the presentation. In order to be clear what level of detail is expected, the software architect should use phase zero, when expectations are set, as an opportunity to ask for clarification.

Step 4 – Identify architectural approaches

By the time this step takes place, the participants should be familiar with the design concepts used in the architecture. This includes software architecture patterns, reference architectures, tactics, and any externally developed software. This information was available in *Phase 0* when the architecture documentation was reviewed, as well as in the prior step (*Step 3 – Present the architecture*) when the architecture was presented. This step is to simply record the design concepts used so that the list can be used in a subsequent step for analysis.

Step 5 – Generate the quality attribute utility tree

Quality attribute scenarios can be represented in a **utility tree**, which represents the usefulness (utility) of the system. Utility trees help participants understand the quality attribute scenarios.

The utility tree is a set of detailed statements about the quality attributes and scenarios that are important to the software system. Each entry in the tree begins with the quality attribute itself (for example, maintainability, usability, availability, performance, or security), followed by a subcategory that breaks it down with more detail, followed by a quality attribute scenario.

For example, under a software quality attribute such as Security, we may have multiple subcategories ("Authentication" and "Confidentiality"). Each subcategory will have one or more quality attribute scenarios:

Quality attribute	**Subcategory**	**Scenario**
Security		
	Authentication	
		User passwords will be hashed using the bcrypt hashing function.
	Confidentiality	
		A user playing the role of a customer-service representative will only be able to view the last four digits of a customer's social security number.
		A user playing the role of a customer-service manager will be able to view a customer's entire social security number.
Performance		
	Etc.	

In addition to identifying the quality attribute scenarios, the project decision makers should prioritize them. As with the SAAM, a voting scheme can be used to allow participants to prioritize the scenarios.

Step 6 – Analyze architectural approaches

The quality attributes that were determined to be of the highest priority in *Step 5 – Generate the quality attribute utility tree*, are analyzed, one by one, by the evaluation team in this step. The software architect should be able to explain how the architecture can satisfy each one.

The evaluation team looks to identify, document, and ask about the architectural decisions that were made to support the scenario. Any issues, risks, or tradeoffs with the architectural decisions are raised and documented. The goal of the team is to match architectural decisions with quality attribute scenarios and determine whether the architecture and those architectural decisions can support the scenarios.

By the completion of this step, the team should have a good understanding of the overall architecture, the design decisions that were made, the rationale behind the decisions, and how the architecture supports the main goals of the system. The team should also now be aware of any risks, issues, and tradeoffs that may exist. The completion of this step signifies the end of *Phase 1*.

Phase 2 – Evaluation (continued)

Phase 2 is a continuation of the architecture evaluation. It is normally scheduled to occur after a short hiatus (for example, one week) after the completion of *Phase 1*. Phase two involves a greater number of participants as compared with *Phase 1*. In addition to the evaluation team and the project decision makers, it is now time for the invited stakeholders to join the evaluation and participate.

This phase should begin with a repeat of *Step 1 – Present the ATAM*, if any of the new participants are unfamiliar with the approach. The evaluation team leader should also summarize what was accomplished in *Phase 1*.

Phase 2 consists of the following three steps:

7. Brainstorm and prioritize scenarios
8. Analyze architectural approaches
9. Present results

Step 7 – Brainstorm and prioritize scenarios

In this step, all of the stakeholders are asked to brainstorm scenarios. They should be encouraged to provide scenarios that are from their perspective and that are important to the success of their roles. Having a variety of stakeholders is helpful to get a diverse set of scenarios.

When there are enough scenarios, the group should look them over to see whether any of them can be removed or merged with others because of their similarities with other scenarios. The scenarios should then be prioritized by the stakeholders by voting in order to determine the most important scenarios.

Once a list of scenarios is determined, it should be compared with the scenarios that the project decision makers came up with for the utility tree in *Step 5 – Generate the quality attribute utility tree*. While the utility tree shows what the software architect and other project decision makers saw as the goals and architectural drivers of the system, this step allows the stakeholders to show what is important to them.

If the two prioritized lists of quality attribute scenarios are similar, it is an indication that the software architect and the stakeholders are in alignment. If any important quality attribute scenarios are uncovered that had not been considered previously, some additional work will be necessary. The level of risk is dependent on the nature and size of the needed changes.

Step 8 – Analyze architectural approaches

Similar to *Step 6 – Analyze architectural approaches*, the software architect describes to the group how the list of scenarios created by the stakeholders can be realized by the architectural approaches that have been taken with the system. The evaluation team can raise any issues, risks, and tradeoffs they see with the architectural approaches. The team should be able to determine whether the scenarios can be achieved by the architecture.

Step 9 – Present results

In the final step of the ATAM, any risks that were uncovered during the evaluation should be related to one or more of the business drivers identified in *Step 2 – Present the business drivers*. Project management will now be aware of the risks and how they relate to the goals of the system and will be in a position to manage those risks.

A presentation is given to the stakeholders that summarizes all of the findings from the evaluation. The output of the process includes the architectural approaches, the prioritized list of scenarios generated by the stakeholders, the utility tree, and documentation regarding the issues, risks, and tradeoffs identified. This output is presented and delivered by the evaluation team to the project decision makers and the stakeholders who participated.

Phase 3 – Follow-up

The evaluation team produces and delivers the final evaluation report in this phase. A common timeframe for this phase is one week, but it could be shorter or longer. The report may be given to various stakeholders for review, but once it is complete, the evaluation team delivers it to the individual who commissioned the review.

Active design review (ADR)

An **active design review** (**ADR**) is most suited for architecture designs that are in progress. This type of architecture review is more focused on reviewing individual sections of the architecture at a time, rather than performing a general review. The process involves identifying design issues and other faults with the architecture so that they may be corrected as quickly and early in the overall design process as possible.

One of the main premises of ADR is that there is too much information involved with reviewing an entire architecture at once and not enough time to do it properly. Many reviewers may not be familiar with the goals and details of every part of the design. As a result, no single part of the design ends up getting a complete evaluation. In addition, with more conventional review processes, there may not be enough one-on-one interaction between the reviewer and the designer.

ADR attempts to address these deficiencies by changing the focus from a more general review of the entire architecture to a series of more focused reviews. Questionnaires are used to provide more opportunities for interaction between reviewers and designers and to keep reviewers engaged.

ADR steps

There are five steps to the ADR process:

1. Prepare the documentation for review
2. Identify the specialized reviews
3. Identify the reviewers needed
4. Design the questionnaires
5. Conduct the review

We will now take a look at each of these steps in detail.

Step 1 – Prepare the documentation for review

In this step of ADR, preparation takes place for the review. This includes preparing the documentation for review and listing assumptions that are being made about the portion of the architecture being reviewed. The assumptions need to be made clear so that reviewers will be aware of them. These assumptions include any items that the software architect (or other designers) think will never change or are highly unlikely to change. In addition, any incorrect usage assumptions should also be provided. These are assumptions that the software architect (or other designers) deem to be an incorrect usage of the module and therefore should not take place.

Step 2 – Identify the specialized reviews

In this step, we identify the specific properties of the design that we want the focus to be placed on by the reviewers. Doing this gives reviewers a clear focus and responsibility for the specialized review that they will be conducting. For example, we may want an individual reviewer to be focused on one or more specific quality attributes.

Step 3 – Identify the reviewers needed

In this ADR step, the reviewers for the part of the design being reviewed are identified. We want reviewers to focus on the areas they are most suited to review. The goal is to get people with different perspectives and sets of knowledge to participate as reviewers.

Examples of reviewers include development team members who did not work on the part of the architecture being reviewed, technical staff from other projects, users of the system, non-technical reviewers who are specialists or have knowledge related to the software system, reviewers who are external to the organization, and anyone else who may be adept at identifying potential issues with a design.

Step 4 – Design the questionnaires

Questionnaires are designed in this step, which the reviewers will use while evaluating the architecture in the next step. Use of the questionnaires is intended to encourage reviewers to take on an active role and to get them to use the architecture documentation during the review. In addition to questions, questionnaires can contain exercises or other instructions for the reviewers to perform.

Questions/instructions should be phrased in an open, active way to encourage further thought and a more detailed response. For example, rather than asking if the part of the architecture being reviewed is sufficient, the instructions could guide the reviewer to provide an implementation in pseudocode that uses the portion of the architecture to accomplish some task.

Step 5 – Conduct the review

This step of the process is when the review takes place. Reviewers are assigned to the review and then a presentation of the module being reviewed is made. The reviewers then conduct their reviews, including the completion of the questionnaires. Sufficient time must be allotted so that the review can be completed properly.

Once the reviews are complete, a meeting is held between the reviewers and the designers. The designers can read the complete questionnaires and use the meeting as an opportunity to communicate with the reviewers and clarify any questions. The end result of the final step is to modify the architecture artifact, if necessary, based on any points made during the review.

Active reviews of intermediate designs (ARID)

Active reviews of intermediate designs (**ARID**) is an architecture review method that combines ADR with ATAM. This hybrid method takes from the ADR approach the focus of reviewing a software architecture while it is in progress and the emphasis on active reviewer participation. It combines this with the ATAM approach of focusing on quality attribute scenarios. The goal is to provide valuable feedback into the viability of the software architecture and uncover any errors and inadequacies with it.

ARID participant roles

The main participants in the ARID process are the ARID review team (facilitator, scribe, and questioners), the software architect/lead designer, and the reviewers:

- **Facilitator**: The facilitator works with the software architect to prepare for the review meeting and facilitates it when it takes place.
- **Scribe**: The scribe captures the issues and results of the review meeting.
- **Questioners**: One or more questioners raise issues, ask questions, and assist with creating scenarios during the review meeting.
- **Software architect/lead designer**: The software architect (or designer) is the person responsible for the design being reviewed. This person is responsible for preparing and presenting the design as well as participating in the other steps.
- **Reviewers**: The reviewers are the individuals who will be performing the review. They consist of stakeholders who have a vested interest in the architecture and the software application.

ARID phases

There are two phases involved in the ARID process, which consist of nine steps in all. The two phases are the *Step 1 – Pre-meeting* and *Step 2 – Review meeting* phases.

Phase 1 – Pre-meeting

The first phase is a meeting to prepare for the actual review. For a software architecture review, this meeting typically takes place between the software architect and the review facilitator. If someone other than the software architect is responsible for the design of the portion of the architecture being reviewed, this person should join the meeting with the review facilitator. The pre-meeting consists of the following steps:

1. Identify reviewers
2. Prepare the design presentation
3. Prepare the seed scenarios
4. Prepare for the review meeting

Step 1 – Identify reviewers

In this step in the ARID approach, the software architect and review facilitator meet to identify the group of people who will attend the review meeting. Management may also be involved with this step to help identify available resources.

Step 2 – Prepare the design presentation

During the actual review meeting, the software architect will present the design and any of the relevant documentation related to it. In this step, the software architect gives a preliminary version of the presentation to the review facilitator. This allows the software architect to practice the presentation and receive feedback from the review facilitator that may help to improve the presentation.

Step 3 – Prepare the seed scenarios

In this step of review preparation, the software architect and review facilitator work together to come up with *seed scenarios*, or a sample set of scenarios that the reviewers can use during the review.

Step 4 – Prepare for the review meeting

This step is used for any other tasks related to the preparation of the review meeting. It can be used to identify the materials that will be distributed to all of the reviewers, such as the architecture documentation, seed scenarios, questionnaires, and review agenda. A date, time, and location for the review meeting must be selected and invitations must be sent out.

Phase 2 – Review meeting

The second ARID phase is devoted to the review meeting. It consists of the following steps:

5. Present the ARID method
6. Present the design
7. Brainstorm and prioritize scenarios
8. Perform the review
9. Present conclusions

Step 5 – Present the ARID method

At the start of the review meeting, the review facilitator should present the ARID method to all of the participants. This step is similar to a step we covered in the ATAM in which the ATAM is presented to participants. As was the case in that step, if everyone who is participating is already familiar with the ARID method, this step could be skipped. However, if anyone is not familiar with ARID or needs a refresher, it should be presented.

Step 6 – Present the design

After the review facilitator has presented the ARID method, the software architect presents the architecture design. Questions regarding the design rationale or comments about alternative solutions should be avoided. The design facilitator can help to keep the meeting in line with its goals. The purpose of the review is to determine whether the designed architecture is usable, so factual questions for clarification and pointing out issues that should be addressed are the types of feedback that should be encouraged. A person playing the role of a scribe should take notes regarding any questions and issues that are raised.

Step 7 – Brainstorm and prioritize scenarios

During this step of the process, the participants brainstorm to come up with scenarios for the software system that will use the designed architecture. The seed scenarios that were created in phase one are included with the scenarios that are created in this step to form the available choices.

As with ATAM, participants should be encouraged to provide scenarios that are important to them. The process works best when there are a variety of stakeholders to ensure that different perspectives are considered.

The group can then analyze and prioritize the scenarios. It may make sense to combine some of the scenarios or identify some as being duplicates. The review group can vote on which scenarios are the most important. The scenarios that are prioritized as being the most important essentially define what makes the architecture usable.

Step 8 – Perform the review

The reviewers use the scenarios to determine whether the architecture solves the problem that is presented. Real code or pseudocode can be written to test the scenario. When the group feels that a conclusion can be reached (or time runs out), the review ends.

Step 9 – Present conclusions

With the review complete, the group should be able to draw conclusions as to whether or not the architecture is suitable for the key scenarios. Any issues with the architecture can be reviewed so that the architecture can be refactored to correct any problems.

Summary

Documenting a software architecture is an important step in delivering an architecture. The documentation communicates the architecture to others, assists the development team, educates team members, provides input for architecture reviews, and allows the reuse of architectural knowledge.

Architecture views, which are representations of an architecture, allow a software architect to communicate their architecture in a manageable and understandable way. There is a cost associated with creating and maintaining views though, so while we never want the documentation to be insufficient, we do not want to spend time working on views that are not needed. In this chapter, you were provided with an overview of the UML, one of the more widely used modeling languages. You learned about structural and behavioral modeling.

Reviewing a software architecture is important to determine whether the architecture will meet the needs of the system. This chapter provided details on several different architecture review methods.

In the next chapter, we will gain an understanding of what software architects need to know about DevOps, including its values and practices. We will learn how continuous integration, continuous delivery, and continuous deployment allow an organization to release software changes quickly and reliably.