Software Architecture

COURSE NOTES

Version 0.1.0

TABLE OF CONTENTS

COURSE OVERVIEW

This is the third course of the Software Design and Architecture specialization brought to you in partnership by Coursera and the University of Alberta. The first two courses of this specialization were Object-Oriented Design and Design Patterns. The two courses provide a foundation for this **Software Architecture** course, although you can take each course separately.

Software Architecture covers how a software system is constructed at the highest level. Development teams will decide how a software system will be broken down into components that work together. This course will show you how developers use UML to document their architectural designs and how architecture can be understood from multiple perspectives and through different UML diagrams.

Next, you will learn about different software architectures, which include when and how they are used, their advantages and disadvantages, and how they are represented visually. Finally, this course will cover how architectures are analyzed and evaluated.

Software architecture is tied to implementation languages, deployment environments, the structure of the development team, the needs of the business and the development cycle, opportunities for product lines, and the intent of the software itself.

Upon completion of this course, you will be able to:

1) Use Unified Modelling Language (UML) to document architectural design.
2) Explain how architecture can be understood from multiple perspectives and through different UML diagrams.
3) Explain different software architectures, including:
    - when and how they are used
    - their advantages and disadvantages
    - how to represent them visually
4) Analyze and evaluate system architectures.
5) Explain how software architecture is tied to:
    - implementation languages
    - deployment environments
    - the structure of the development team
    - the needs of the business and development cycle
    - opportunities for product lines
    - the intent of the software

MODULE 1: UML ARCHITECTURE DIAGRAMS

Upon completion of this module, you will be able to:

(a) Explain the architecture overview and process.
(b) Describe Kruchten's 4+1 View Model.
(c) Describe Component diagrams.
(d) Describe Package diagrams.
(e) Describe Deployment diagrams.
(f) Describe Activity diagrams.

Architecture Overview and Process

Software architecture is the fundamental design of an entire software system. It defines what elements are included in the system, what function each element has, and how each element relates to one another. It is the big picture or overall structure of the whole system—how everything works together. It follows that to design a software system, a software architect has to take many factors into consideration:

- the purpose of the system,
- the audience or users of the system,
- the qualities that are of most importance to users, and
- where the system will run.

Software architecture is important, particularly for large systems. If there is a clear design of the overall system from the start, there is a solid basis for developers to follow. Each developer will then know what needs to be implemented and how things are related to meet desired needs efficiently. This avoids conflicts, duplication, and ad hoc unnecessary work.

Some advantages of software architecture include:

- higher **productivity** for the software team, as a well-defined structure helps to coordinate work, implement individual features, or guide discussions on potential issues
- improved **evolution** for the software, since design principles are applied to make changes easier to accomplish or defects easier to find
- enhanced **quality** in the software by carefully considering the needs and perspectives of all the stakeholders

Software architecture helps make a system easier to maintain, reuse, and adapt.

In order to develop software architecture, architects must take into account the **stakeholders** of the system. Stakeholders are the people who have an interest in the software at hand. They either use the system or benefit from it in some way.

Software architecture typically have the following stakeholders. Each will have their own perspective.

Stakeholder	Description
Software developers	Software architecture helps developers create and evolve software by providing strong direction and organization on what needs to be done.
Project Managers	Software architecture provides useful information to project managers to help them identify possible risks and to manage the project successfully. Software architecture helps project managers to understand task dependencies and impacts of change and to coordinate work assignments.
Clients	Clients make important decisions about the system, like its funding. Software architecture establishes a basis for communication with clients, so they understand what they are paying for and that their needs are met.
End users	Users may not care how the software is actually designed, but they do care that it “works well” for them.

One important way software architecture is presented is through UML diagrams.

Kruchten’s 4+1 View Model

Multiple perspectives are necessary to capture the complete behaviour and development of a software system. There are several important considerations.

One consideration is the functionality of the software. Functionality involves what a system does to satisfy the purpose the client desires.

Focusing on this functionality and the needed objects leads to a perspective called the **logical view**.

Another consideration is how well the software executes, using characteristics like the efficiency of the system or the interaction of subprocesses. These characteristics affect the performance and scalability of the system. Focusing on the processes implemented by the objects in the logical view leads to a perspective called the **process view**.

Software can also involve the **development view**. This perspective focuses on implementation considerations such as the hierarchical structure of the software. The programming languages of the system will heavily influence this structure and therefore places constraints upon development.

Another perspective of the software can be seen through the **physical view**. The software will have physical components that interact and need to be deployed. The interaction between these different elements and their deployment will affect how the system works.

All four views share the purpose or desired capabilities of the software as defined by the client. **Scenarios** outline use cases or tasks required by the end users, which provides context to help to detail the four views.

Together, logical, process, development, and physical views, along with scenarios form Philippe **Kruchten's 4+1 View Model**. This model is a way of understanding the key considerations or important perspectives that need to be addressed in software architecture.

Let us examine each of these views individually.

Logical View

The logical view, which focuses on the functional requirements of a system, usually involves the objects of the system. From these objects, a UML class diagram can be created to illustrate the logical view.

A class diagram establishes the vocabulary of the problem and resulting system. By defining all of the classes, their attributes, and their behaviours it becomes easy to understand the key abstractions and terminology. Class diagrams are also useful for specifying database schemes. The class diagram makes it easier to see how classes interact and how data should relate to each other in a database.

DID YOU KNOW?

Some of the most effective UML diagrams related to the logical view of a system are the class diagram and the state diagram. Both the class diagram and the state diagram focus on the classes and objects of a system.

Process View

The process view focuses on achieving non-functional requirements. These are the requirements that specify the desired qualities for the system, which include quality attributes such as performance and availability. The process view also presents processes that correspond to the objects in the logical view.

DID YOU KNOW?

Some of the most effective UML diagrams related to the process view of a system are the activity diagram and the sequence diagram. The activity diagram can illustrate the processes or activities for a system. The sequence diagram shows how objects interact with one another, which involves how methods are executed and in what order.

Development View

The development view describes the hierarchical software structure. It also considers elements such as programming language, libraries, and toolsets. It is concerned with the details of software development and what is involved to support that. This extends to management details such as scheduling, budgets, and work assignments. Essentially, the development view covers the hierarchical software structure and project management.

Physical View

The physical view handles how elements in the logical, process, and development views must be mapped to different nodes or hardware for running the system.

DID YOU KNOW?

One of the most effective UML diagrams related to the physical view of a system is the deployment diagram. It can express how the pieces of a system are deployed onto hardware or execution environments.

Scenarios

Scenarios align with the use cases or user tasks of a system and show how the four other views work together.

For each scenario, there is a **script** that describes the sequence of interactions between objects and processes. This involves the key objects defined in the logical view, the processes described in the process view, the hierarchy identified in the development view, and the different nodes specified in the physical view. Scenarios relate these elements to provide a complete picture.

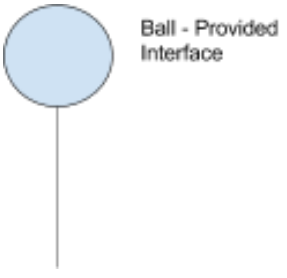
None of the views are fully independent of each other, with elements of some views connected to others. The 4+1 view model can be molded to fit many situations to understand the architecture of a software system. Being able to see a complex problem in many different perspectives helps make your software more versatile.

Component Diagrams

UML component diagrams are concerned with the components of a system. **Components** are the independent, encapsulated units within a system. Each component provides an interface for other components to interact with it. Component diagrams are used to visualize how a system's pieces interact and what relationships they have among them.

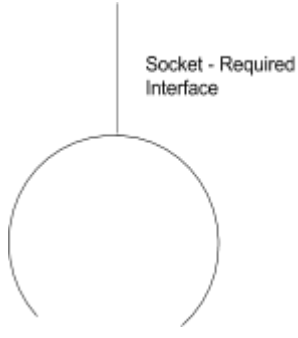
Component diagrams are different from most other diagrams, as they show high-level structure and not details like attributes and methods. They are purely focused on components and their interactions with each other.

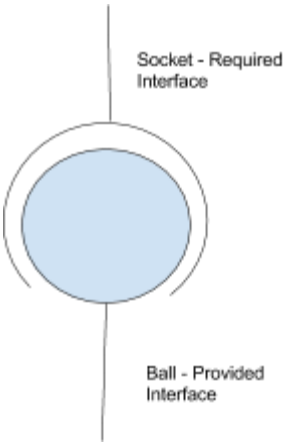
Component diagrams are a static view of the software system, and depict the system design at a specific point in its development and evolution. The basis of component diagrams focuses on the components and their relationships. Each component in a diagram has a very specific relationship to the other components through the interface it provides.



Component diagrams have **ball connectors**, which represent a provided interface. A provided interface shows that a component offers an interface for others to interact with it. The provided interface means that client and consumer components have a way of communicating with that component.

Component diagrams also have **socket connectors** that display a required interface. The required interface is essential to the component diagram, to show that a component expects a certain interface. This required interface is to be satisfied or provided by some other component.

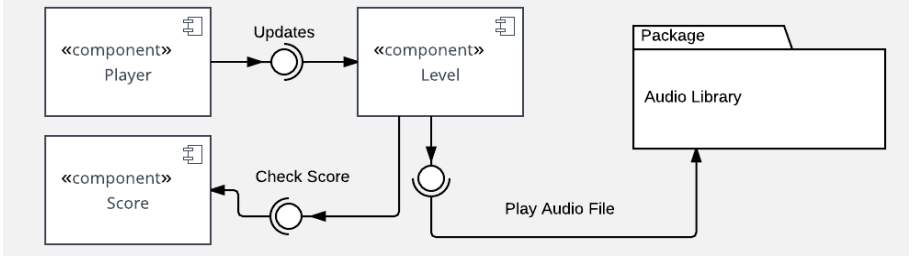




Finally, component diagrams can illustrate an assembly relationship. An assembly relationship occurs when one component's provided interface matches another component's required interface. The provided interface is depicted by a ball, and the required interface is depicted by a socket.

To build a component diagram, first, you must identify the main objects used in the system. Next, the relevant libraries for the system need to be identified. Finally, the relationship between these components would need to be identified.

When relevant libraries are identified, this extends to third-party implementation dependencies, which should also be integrated into the diagram where relevant. Below is an example of a component diagram for a video game system.



Component diagrams are especially useful early in the design process, because of its high-level emphasis. They can be drawn at different levels and allows you to focus not only on systems but on subsystems as well.

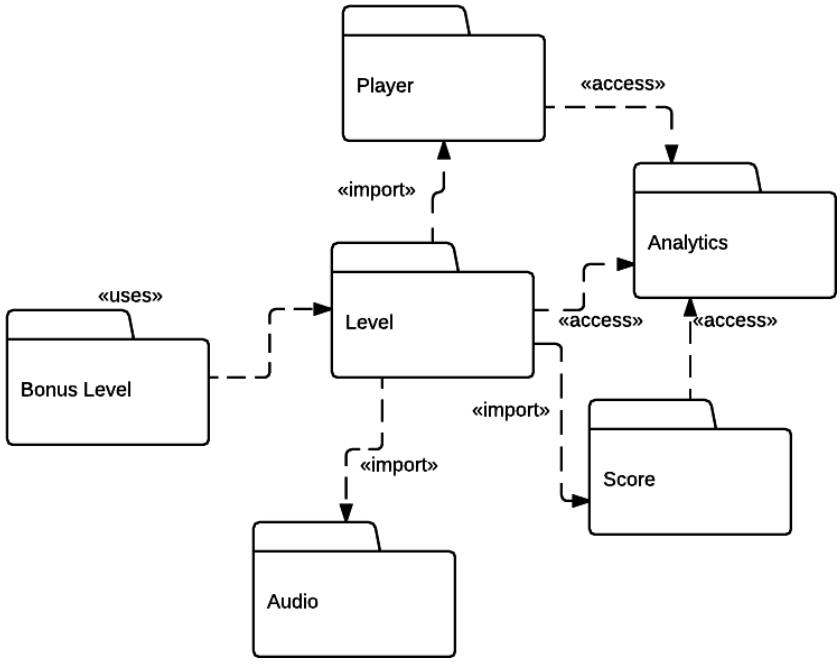
Package Diagrams

Sometimes, classes in object-oriented software are related in some way.

A **package** groups together elements of software that are related. Elements can be related based on data, classes, or user tasks. A package can also define a “namespace” for elements it contains, that is, a package is named and can organize the named elements of software into a separate scope. An element can be uniquely identified in the system by a fully “qualified name” that is based on its own name and the name of the package that the element is in.

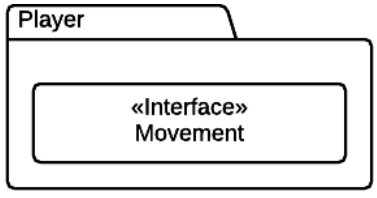
Package diagrams show packages and the dependencies between them. These diagrams can organize a completed system into packages of related **packageable elements**, which could include data, classes, or even other packages. Package diagrams help provide high-level groupings of a system so that it is easy to see how a package contains related elements as well as how different packages depend on each other.

Below is an example of a package diagram for a video game:

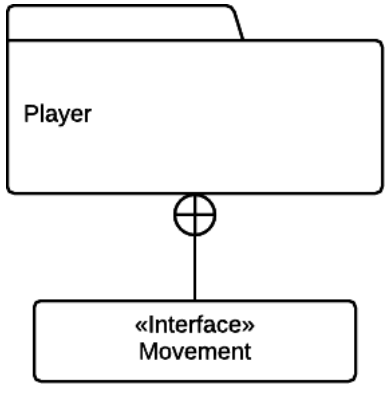


Packages are depicted by tabbed folders. If there are no elements to show in the package, then the package name goes into the centre of the folder. If details are needed, there are two ways this can be expressed.

The elements can be nested within the folder, as below:

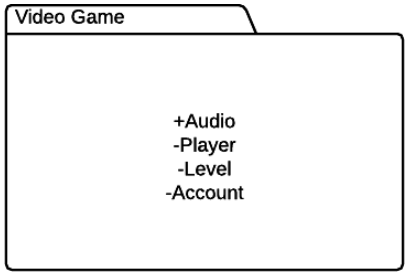


Alternately, this notation can be used:



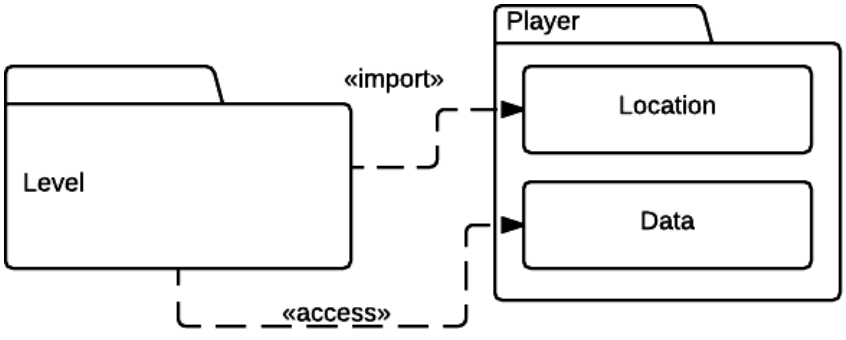
Here, a player package contains the movement interface, but it is drawn below the folder in the diagram. This relationship is a composition, where the whole package has an element as a part.

Contained elements can also be listed in a package by their names. These names can be partially qualified, but they should be unique within the package.

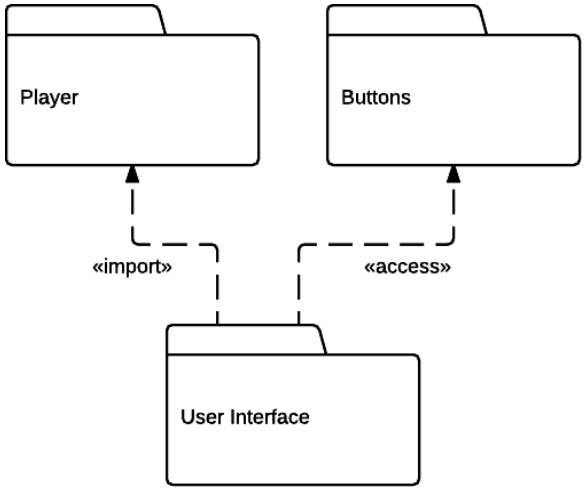


Notice in the above example that plus (+) and minus (-) signs have been used to indicate if elements are public or private. Public elements can be accessed outside of the package by their fully qualified names. Relationships are denoted through dotted-line arrows.

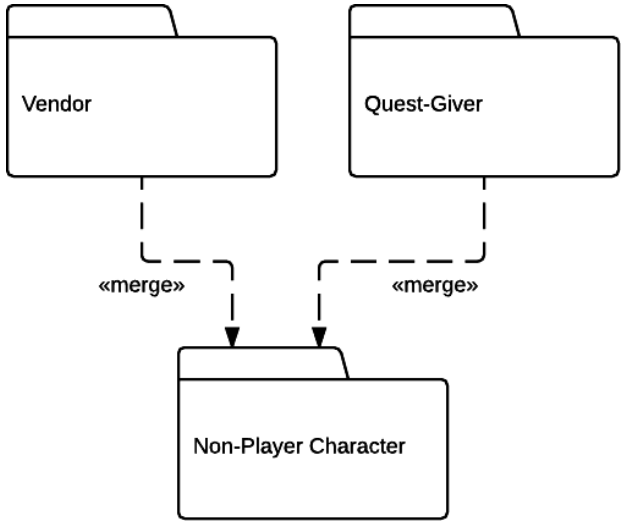
A package can import an element from another package, as in the example below where the Level package is importing Location from the Player package. This brings the name Location into the Level namespace, particularly for convenient use by elements in Level. As the import tag is public, the name is visible for further import through the Level package. If a tag was private, the name would only be visible by elements within the Level namespace and not further outside.



A package can import the entire contents of other packages, as in the example below, where the User Interface package is doing a public import of the Player package, as indicated by the import tag, essentially doing public imports of the visible names of elements in Player. The User Interface package is doing a private import of the Buttons package, as indicated by the access tag, essentially doing private imports of the visible names of elements in Buttons. These names are not made further known outside the User Interface namespace.



Packages can also be merged, as in the diagram below. Merging typically occurs when two packages or concepts need to come together into one. This is a use of generalization that allows different definitions to be provided for the same concept.



The first diagram shown at the beginning of this section shows a complete package diagram for a video game system. Packages such as player, level, score, and analytics on player habits are included. These packages mostly interact with each other, meaning that they can publicly import each other. Note that the analytics package, however, requires some communication with an online server to store data. This package is available by private import.

The bonus relationship is also special because it has a uses relationship with a normal level. The uses relationship implies that the bonus level requires a normal level for its full implementation. If there is no definition for level, there cannot be a definition for bonus level.

Package diagrams can be created at any stage of development. They can also adapt and change with the latest version of software being worked on. Package diagrams are particularly useful for technical designers because they allow them to see the dependencies and relations between groups of related elements.

Deployment Diagram

UML deployment diagrams are used to visualize the deployment details of a software system. The diagrams include more than just code, but also separate libraries, an installer, configuration files, and many other pieces. In order for software to be ready to run, it is necessary to understand all the files and executables involved and the environments where they reside.

The deployment environment, or deployment target, can be very specific and involve particular hardware devices. It can also be very general and involve supported operating systems. Details in a deployment diagram change accordingly. For example, software developed for Linux, MacOS or Windows may have differences from one another.

Deployment diagrams deal with **artifacts**. Artifacts are a physical result of the development process. Artifacts for a video game might include things like an executable to run the game, an installer to install the game, audio libraries for sound, and multimedia assets. These are created as outcomes of producing the system and are the final pieces to be put together.

There are two different types of deployment diagrams:

- **specification-level diagrams**
- **instance level diagrams**

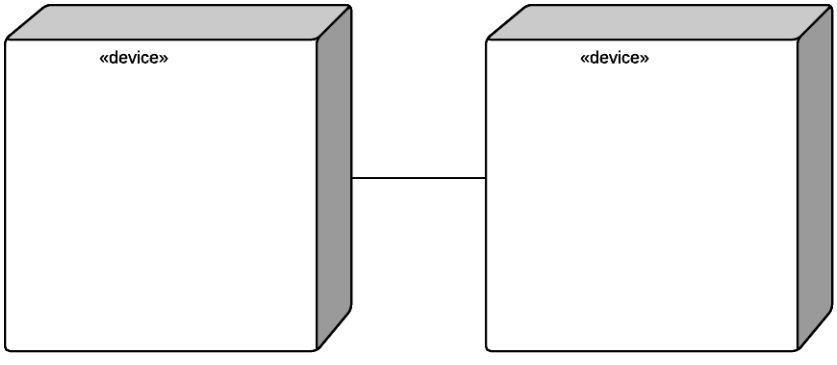
A specification-level diagram provides an overview of artifacts and deployment targets, without referencing specific details like machine names. It focuses on a general overview of your deployment rather than the specifics.

An instance-level diagram is a more specific approach that maps specific artifacts to specific deployment targets. They can identify specific machines and hardware devices. This approach is usually used to highlight the differences in deployments among development, staging, and release builds.

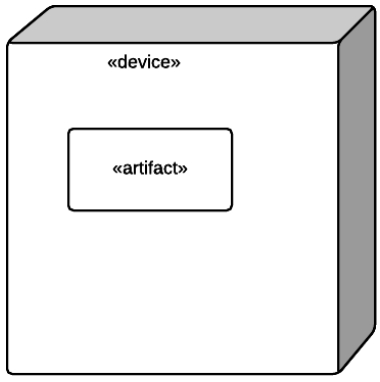
When creating deployment diagrams, it is important to use the correct notation for the various elements. One of the most important aspects of the deployment diagram is the node. **Nodes** are deployment targets that contain artifacts available for execution. In a deployment diagram, they look like 3D boxes.

Hardware devices are also displayed as 3D boxes, only they have a “device” tag on them to differentiate them.

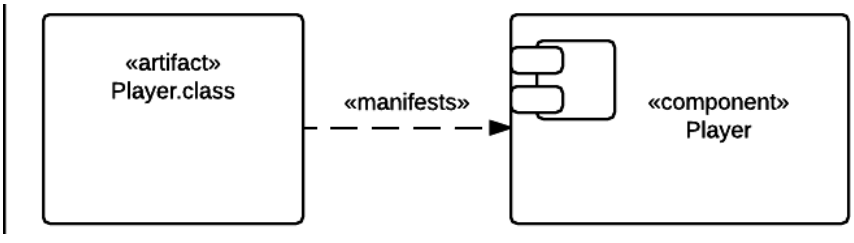
Relationships can be represented a few different ways in deployment diagrams. A solid line between two nodes shows a relationship between deployment targets. The line shows that the two nodes have a communication path between them. This relationship typically identifies a particular communication protocol.



If an artifact is drawn inside a node box, this shows that an artifact is deployed to a node. This also means that the artifact cannot function without this deployment target.

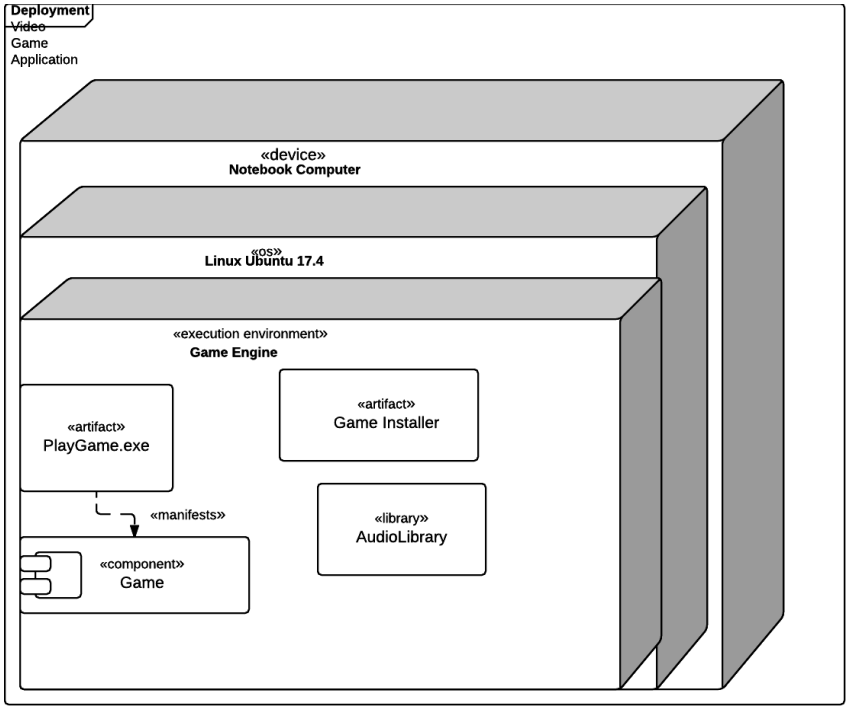


Manifestation is a relationship where an artifact is a physical realization of a software component. It can be represented with a “manifests” indicator, as below.



Here, the Player.class manifests the Player component, which is the encapsulated unit that contains all of the functionality of a player.

Below is a specification-level deployment diagram for a simple video game. This diagram provides a general and concise idea of how to deploy the application without providing too much detail.



In a deployment diagram, there is a distinct hierarchy of deployment targets. This hierarchy is very important. You should start from the highest level of your deployment information, from application name down to device and operating system. In this example, the application, the device, the operating system, and the execution environment are all listed. When you look at each of these nodes, it is clear what environment is necessary to run the application.

Note that only the most important artifacts are represented in this diagram so that it is kept at a high level. Remember, deployment diagrams provide a high-level overview of artifacts, libraries, main components, machines, and devices that your application needs to run.

Deployment diagrams help provide consistency and organization to deployments, which helps avoid system failures. The diagrams also help keep track of the files and executables needed to deploy and run the software. The diagram can be on an instance level specific to the deployment machines you are using, or it can be general for a range of execution environments.

Activity Diagram

A **UML activity diagram** allows the representation of the control flow from activity to another in a software system. It captures the dynamic behaviour of the system and allows the mapping of branching into alternative flows.

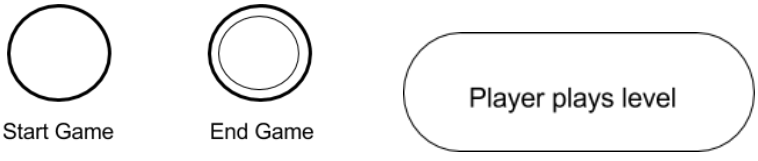
Activities are actions that further the flow of execution in a system. They are actions that when completed cause another action to execute. For example, an action can alter or create new objects. These changes or actions can drive your application forward.

In order to create an activity diagram, you must:

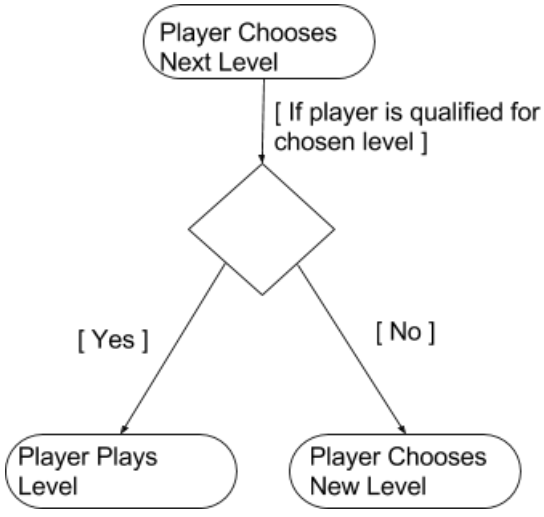
1. Identify the activities.
2. Identify the respective conditions of the system's activities.

Once activities and respective conditions have been determined, the activity diagram can be created. There are a few major parts to an activity diagram.

There are start and end notes that look like labelled circles. These circles are where the diagram must begin. They show the starting activity that initializes the control flow of the application. The end node shows the final activity of the diagram. Intermediate activities are shaped like an oval, and they describe all of the activities that change the game state before the game ends.

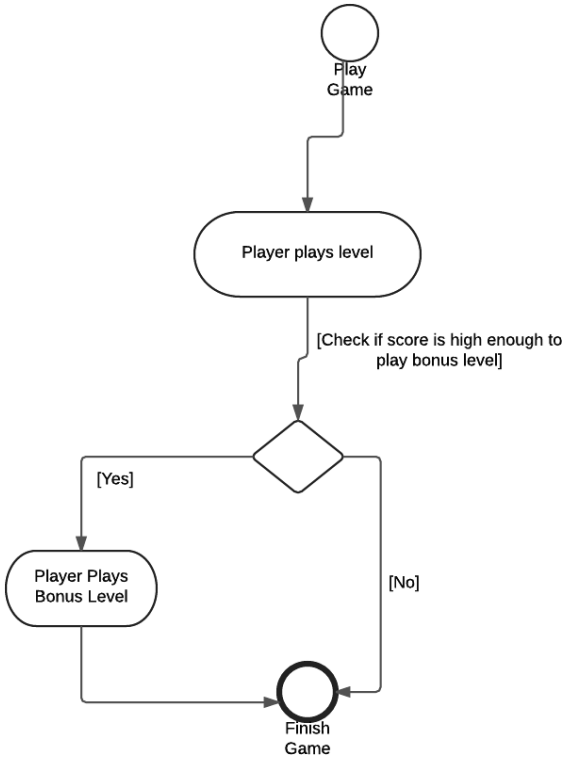


Additionally, activity diagrams can include decision nodes. These are diamonds that have an activity leading into it, and there is the possibility of two alternative outcomes as the next activity. The choice of outcome depends on how the condition on the decision node evaluates. An example of a decision node can be seen below.



All essential activities must be included in a diagram as well as the conditions.

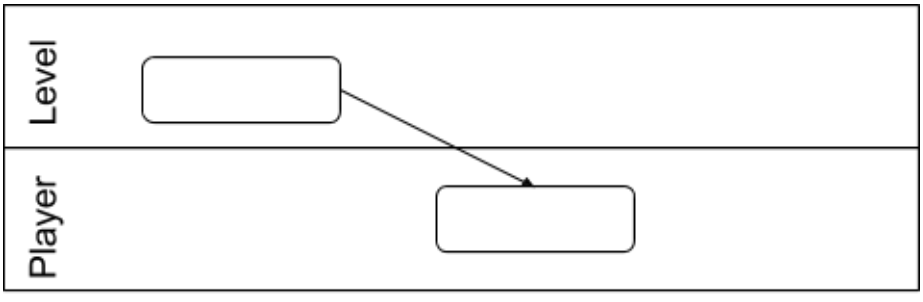
Below is an example of an activity diagram that illustrates a video game.



In this example, the activity diagram begins by playing the game. The player then plays the level activity. When the level is complete, the player encounters the first condition. If the score is high enough, then the player may move on to the bonus level. If the player cannot play the bonus level, then the game is ended, as denoted by a circle.

Activity diagrams allow the mapping of concurrent activities that happen in parallel. A fork in the flow moves into parallel flows, so activities can happen at the same time. Parallel flows can join into a single flow. For example, if a level ends in a video game, the soundtrack flow will end at the same time as the play flow. To denote this, activity diagrams can use a separate **swimlane** for each flow.

Swimlanes can also divide activities up into different categories, such as where it occurs or the user role involved. See the example below for separating player activities and level activities.



In this diagram, there is a label for each swimlane. Arrows crossing each lane show how different activities of a system interact across categories.

Activity diagrams allow you to see what activities and conditions should be included in a system. As well, from the diagrams you see the order in which features are encountered while also allowing for alternate flows to be taken into account for the system.

MODULE 2: ARCHITECTURAL STYLES

Upon completion of this module, you will be able to:

(a) Explain important software architectures, their key characteristics, and how they are used in practice, which include:

- Language-based systems
- Repository-based systems
- Layered systems
- Interpreter-based systems
- Dataflow systems
- Implicit invocation systems
- Process control systems

This module will explain important software architectures, their key characteristics, and how they are used in practice. In particular, we will examine language-based systems, repository-based systems, layered systems, interpreter-based systems, dataflow systems, implicit innovation systems, and feedback control systems.

Language-Based Systems

The programming paradigm of the language selected to implement a system will affect the architectural style of that system. Each programming paradigm has its own set of constructs, principles, and design patterns, and their use shapes the system being created. This section focuses on object-oriented architectural styles, which result from object-oriented programming paradigms.

In previous courses, the following object-oriented principles were explored:

- **Abstraction** that simplifies a concept
- **Encapsulation** that bundles data and functions into a self-contained object, so other objects can interact with it through an exposed interface
- **Decomposition** that breaks a whole into parts
- **Generalization** that allows you to factor out conceptual commonalities

Object-oriented design patterns were also explored. These can fall into the following categories:

- **Creational patterns** that guide the creation of new objects
- **Structural patterns** that describe the relationships between objects and the interactions between classes and subclasses
- **Behavioural patterns** that focus on how objects perform work individually or as a group to accomplish something

These elements lead to an object-oriented architectural style for the system.

Abstract Data Types and Object Oriented Design

Object-oriented design architectural styles are focused on the data. When modelling a system in object-oriented design, begin by looking at the different kinds of data handled by the system to see how the system can be broken down into **abstract data types**. An abstract data type can be represented as a class that you define to organize data attributes in a meaningful way, so related attributes are grouped together along with their associated methods. Encapsulation restricts access to the data and dictates what can be done with it.

Object-oriented refers to a system composed of objects where each object is an instance of a class. In other words, the type of each object is its class.

Objects may interact with each other through the use of their methods. The object-oriented paradigm allows for inheritance among abstract data types. This means that one abstract type can be declared an extension of another type.

These classes themselves form a language-based architecture that arises from basic object-oriented principles. The classes within the system will determine the overall structure of the system. In other words, the overall object-oriented architectural style of a system directly follows from the fact that an object-oriented approach was used in development.

Some problems are well suited to an object-oriented architectural style. However, not all situations will have easily identifiable classes. In some situations, another design choice may be better suited for the problem at hand.

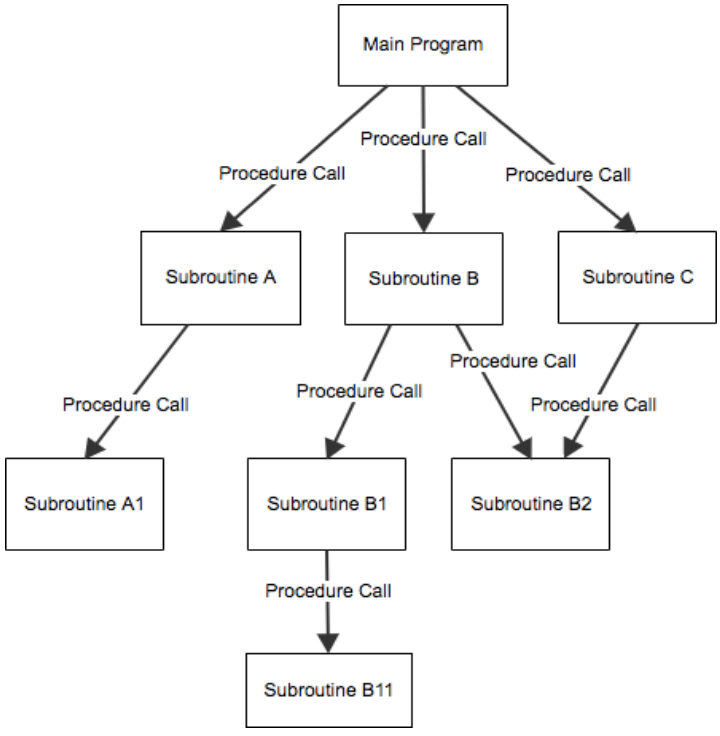
> ***Instructor's Note:***
>
> *For a review on object-oriented thinking, see the Object-Oriented Design course of Software Design and Architecture specialization.*

Main Program and Subroutine

Main program and subroutine architectural styles are focused on functions. They develop from a procedural programming paradigm. C is an example of a language that follows this paradigm.

Note that for this section, the terms routine, procedure, and function are used interchangeably with the term subroutine.

In a main program and subroutine architectural style, a system is modelled by breaking up the overall functionality of the system into a main program and subroutines



The diagram above represents a main program and subroutine architectural style. In the diagram, subroutines are connected by procedure calls, and they may have nested calls. In nested calls, subroutines may call other subroutines, which may call yet more subroutines, and so on. This means that the structure of the resulting code is not flat, but rather it is hierarchical, so it can be modelled as a directed graph. The structure of the subroutines that build up the system affect the structure of the system as a whole. The subroutines declared in the code are structure as a big “call tree.”

In this paradigm, data is stored as variables. Abstract data types are supported in procedural programming; however, inheritance is not explicitly supported. Under this paradigm, it is not easy to make one abstract type an extension of another. The main focus of this paradigm is therefore on the behaviour of functions and how data moves through those functions. The main program and subroutine architectural style is best suited for computation-focused systems.

Each subroutine may have its own local variables. A subroutine has access to all data within its scope. To access data outside its scope, data may be passed into the subroutines as parameters, by value or by reference, and data may be passed out of a subroutine as a return value.

A principle of this paradigm is “one entry, one exit per subroutine,” which makes the control flow of the subroutine easier to follow.

This architectural style promotes modularity and function reuse, which results in certain advantages. When functions are well defined and do not produce side effects, they are considered like “black boxes.” Given an input, they will always have the same output. Further, library functions are easily integrated into programs.

There are also certain disadvantages from this style. Subroutines may mutate data in unexpected ways. A subroutine may be affected by data changes made by another subroutine during execution. This issue is especially true for global data shared across subroutines. Changes to data can be unpredictable and result in a function receiving an input that it was not expecting, or even an input that it is not capable of handling correctly, which can result in run-time errors.

Procedural programming paradigms are well suited to systems centred around computation, such as spending management programs. Identifying object-oriented components of these kinds of systems can be difficult and result in overly complex solutions. However, for problems where abstract data types with modelling make the solution easier, then object-oriented architectural styles are more well suited.

Repository-Based Systems

Modern software systems rarely operate in isolated environments. As a software developer, any software architecture you create needs to be capable of sharing information between separate components.

A number of issues need to be addressed to do this:

- The state of a component is only temporary while the component is running. This means that any objects created and any amount of data processing will be lost when the component stops running.
- Components may not need all the data that is stored in the files. There needs to be a way to communicate the state of the data between multiple components.

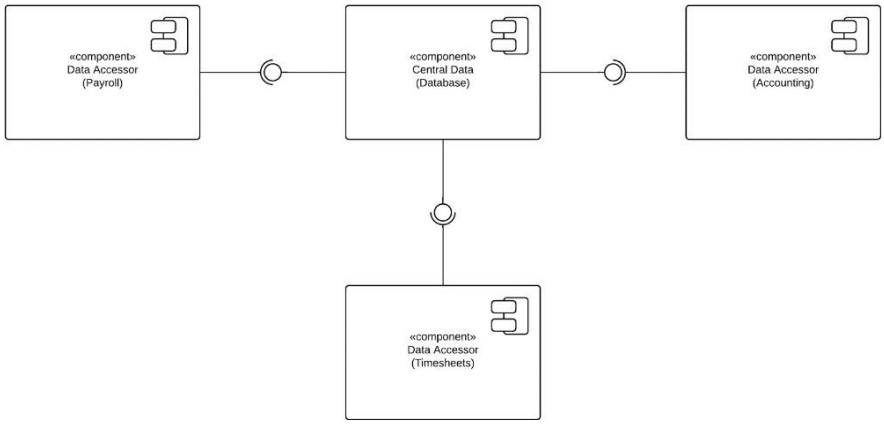
Although it is possible to save information to keep in a file to prevent the loss of data, files are not the best way to transfer data between components.

To solve both of the issues, a **data-centric software architecture** can be used. This architecture allows data to be stored and shared between multiple components, thus increasing the maintainability, reusability, and scalability of the system. This architecture can be achieved by integrating a method of shared data storage, such as a database, into the overall system design.

At the core of a data-centric architecture are two types of components:

- **Central data** is the component used to store and serve data across all components that connect to it.
- **Data accessors** are the components that connect to the central data component. The data accessors make queries and transactions against the information stored in the database.

Below is a diagram of a data-centric software architecture.



Data accessors are separated from one another, and they communicate only to the central data component. The central data shares data by saving the desired information from the current state of the system and serving data when requested.

In order to better understand this system, it is helpful to examine each of these components.

Databases

Central data is often stored in a **database**. Databases ensure several data qualities, which are important to data-centric architectures. These are:

- **Data integrity**. A database ensures that data is accurate and consistent over its lifespan.
- **Data persistence**. A database ensures that data will live on after a process has been terminated—databases can save data from any number of components.

One way that databases store information is by using tables. Relational databases are a type of database that uses tables. Each table represents an abstraction.

Relational databases use **Structured Query Language (SQL)** to query or “ask” the database for information and to perform transactions or “tell” the database to do something. These abilities allow for information to be shared through a database between data accessors.

The management and optimization of queries and transactions can be automated by a **database management system (DBMS)**, so integrating a database into a system is simplified.

In data-centric architectural design, the central data is passive. Central data is primarily concerned with storing and serving information, not with heavy data processing or large amounts of business logic.

Data accessors are any component that connects to a database, which is characterized by its abilities to:

- Share a set of data, while being able to operate independently.
- Communicate with the database through database queries and transactions. Data accessors do not need to interact directly with each other and therefore do not need to know about each other.
- Query the database to obtain shared system information. This is used to get data in order to perform computations.
- Save the new state of the system back into the database using transactions. Data is stored back into the database once the data accessor has finished its processing.

A data accessor contains all the business rules required to perform its functions. This means that this software architecture enables you to separate concerns into different, specialized data accessors. Also, use of the data accessors can be controlled, so an end user only has permission for the ones they need on a day-to-day basis.

Data-centric architecture presents many advantages over a basic object-oriented system, because of the integration of a centralized database. These advantages include:

- Increased support of data integrity, data backup, and data restoration.

- Reduced overhead for data transfer between data accessor, as data accessors do not need to be concerned with talking to one another—the database communicates for them,

- A system that can be easily scaled up, as data accessors are functionally independent, so additional features can be added with having to worry about affecting others.

- Central data components usually “live” on a separate server machine with sufficient disk storage dedicated to the database, which allow for easier management of information.

Data-centric architecture also presents disadvantages. Integrating a database can disadvantage a system in the following ways:

- The system is heavily reliant on the central data component, so if it becomes unavailable or if the data corrupts, then the entire system is affected. Safeguards such as data redundancies that replicate data on separate hard disks can be costly.

- Data accessors are dependent on what gets stored in the database. New data accessors need to build around the existing data schema. Anything that isn’t stored needs to be computed, or if there is no matching column or table for a specific data need, then the database cannot be used.

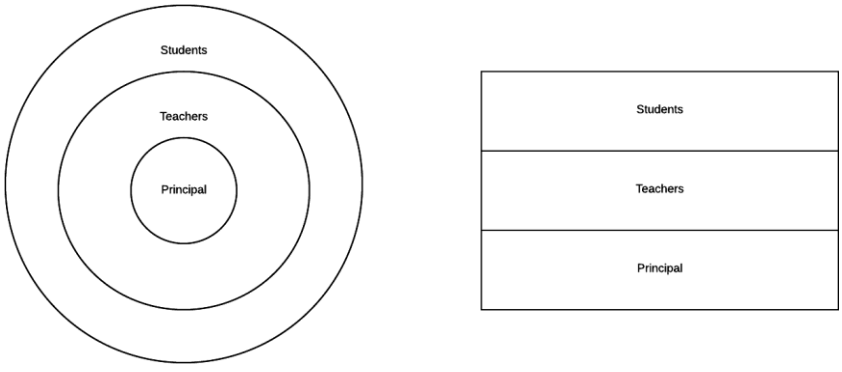
- It is difficult to change the existing data schema, particularly if a large amount of data is already stored. Further, data schema changes will affect data accessors.

In summary, data-centric software architecture is a commonly used design by many companies. They allow large amounts of data to be stored and managed in a central data repository, making a system more stable, reusable, maintainable, and exhibit better performance. They also separate the functionality of data accessors, so it is easier to scale the system. Finally, they also facilitate data sharing between data accessors through database queries and transactions.

As always, the decision to use a data-centric architecture relies on the context of the problem at hand.

Layered Systems

Many organizations are designed by hierarchy. Consider a school, for example, which could be visualized as below:



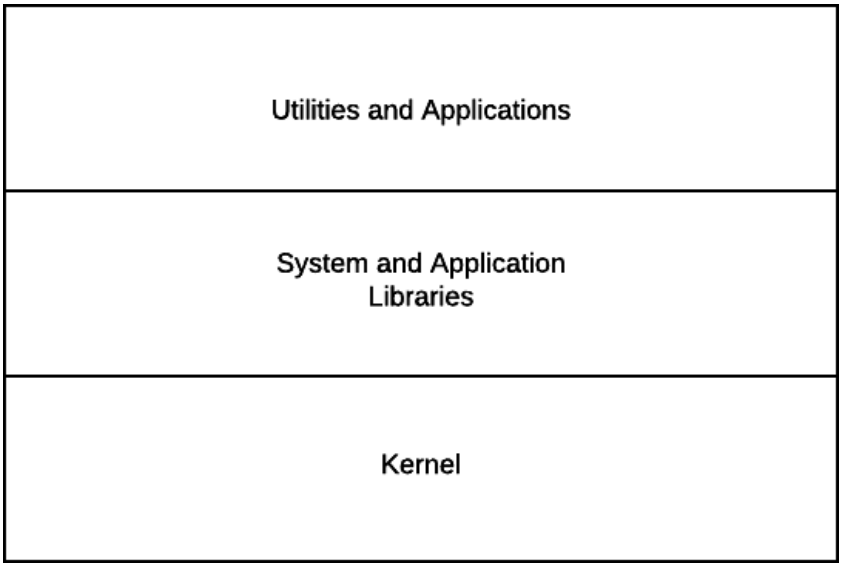
In this diagram, there are three groups, which are the three layers of the organization. People in the adjacent layers interact. So, students and the principal only interact with one layer—the teachers, while teachers interact with two adjacent layers—students and the principal. People in each layer can also interact with each other. So, teachers can also interact with other teachers.

In software, a layer is a collection of components that work together towards a common purpose. The key characteristic of a layered architecture is that the components in a layer only interact with components in their own layer or adjacent layers. When an upper layer interacts with the layer directly below it, it cannot see any of the deeper layers.

Usually, the inner or bottom layer provides services or abstractions for the layer outside or above it. The interfaces provided by the components of a layer should be well defined and driven by the needs of the system.

Layering allows for applying “separation of concerns” into each of the layers. Many layered systems are split into “presentation,” “logic,” and “data” layers.

An operating system for a computer is a common example of a layered system.



The kernel is the core of the operating system. One of its main responsibilities is to interface with the hardware and allocating resources. Above the kernel layer is the system and application libraries layer. This layer provides the higher-level functions for saving files or drawing graphics. Above this layer is the utilities and applications layer, which to most users is the operating system. Utilities are tools included with the operating system, such as command-line programs. Applications are additional programs, often installed by the user for specific needs.

Advantages of layered systems include the fact that:

- Users can perform complex tasks without needing to understand the layers below.
- Different layers can be run at different levels of authorization or privilege. Typically, the top layer is considered the "user space" that does not have authority to allocate system resources or communicate with hardware directly. This "**sandboxing**" provides security and reliability to the kernel.
- Designs will be more loosely coupled, as layered architecture follows the **principle of least knowledge**.

Lower layers typically provide services to the layer above. If a layer is replaced, the provided interface for the layer above must be consistent with the previous implementation.

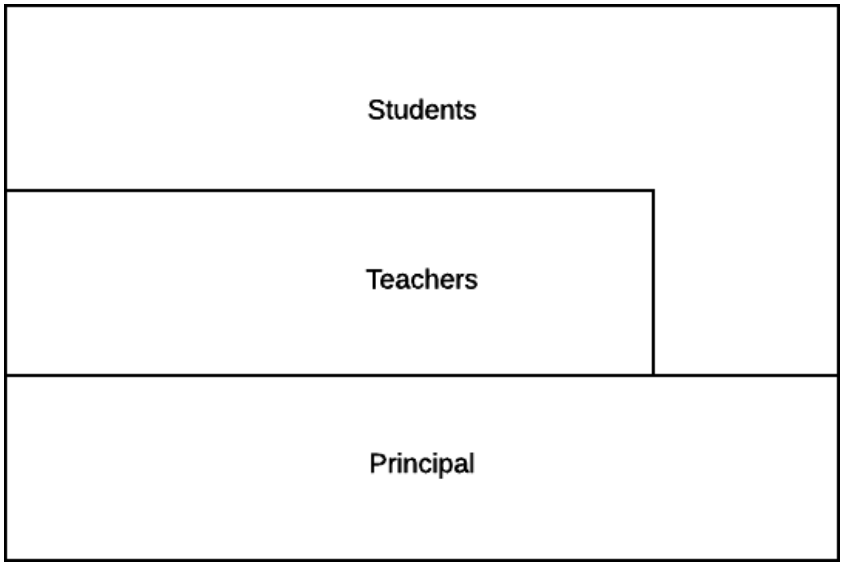
Not every software problem can easily be mapped to a layered architecture. It may not always be obvious how a solution can be made into layers and where the division for software will be for a collection of object-oriented classes.

There are certain trade-offs to be aware of in layered systems. Most notably, enforcing layers has an efficiency trade-off. If only adjacent layers can communicate, then the system will likely have some interactions that pass through one layer to the next, and sometimes information must be shared between layers that are not adjacent. This extra communication adds complexity and uses up processing resources.

Overhead must be balanced against the **separation of concerns** gained from enforcing layers. If most communication is passing directly through a layer, then the design may need to be relaxed, or a different architecture may be better suited to the problem. If only a small amount of communication is passed through, then a layered system may be a good choice.

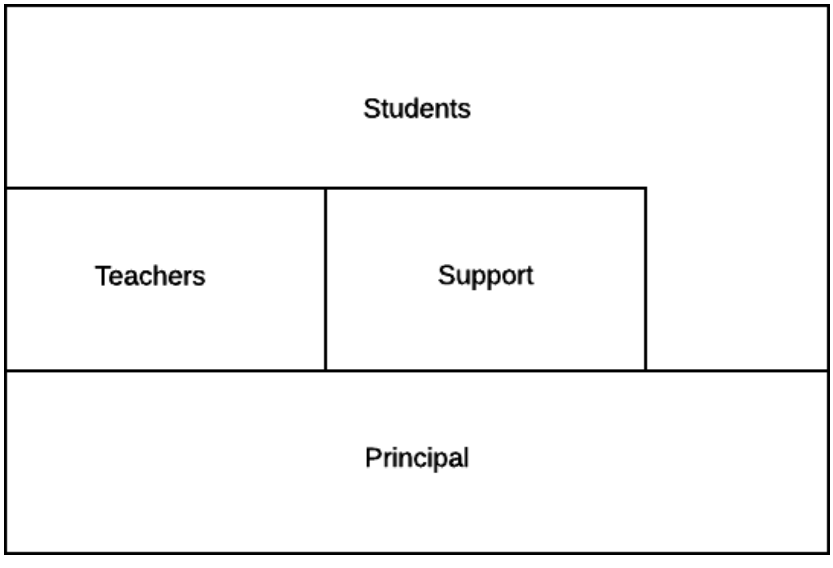
Layered architecture can be relaxed by allowing for passthrough. In a diagram, this is shown as a notch in the layer. Let us return to the example at the beginning of this section—a school organization. Suppose that the principal would like to speak to the students.

The diagram would be reworked as below:



This is another trade-off, however. Although communication has opened up, the organization is now more tightly coupled, and more complex.

Some layered systems may have more than one grouping in a layer.



In summary, layered architecture:

- It can be intuitive and powerful. Many organizations and solutions have layered structures, so it is easy to apply layered architecture where appropriate.
- It supports the separation of concerns, because each layer is a set of components with similar responsibility or purpose.
- It is modular and loosely coupled, as each layer only communications with one or two other layers, allowing for different implementations to be easily swapped.
- It can be adapted so that layering is not always strict, which helps manage design complexity or provide a starting point for structuring the system.

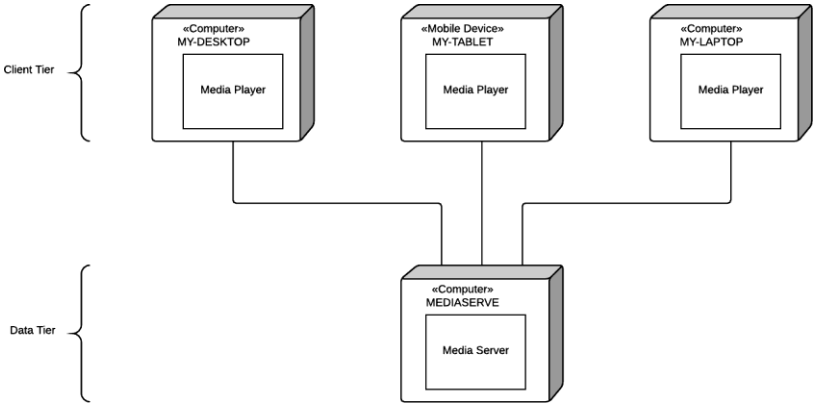
Client Server n-Tier

n-Tier or **multitier** architectures are related to layered architectures. **Tiers** refer to components that are typically on different physical machines. Although the terms “tier” and “layer” are used interchangeably, they are not the same thing. n-Tier or multitier architectures are layered architectures based on tiers.

The number of tiers in an n-Tier architecture can vary, although three-tier and four-tier architectures are common. The relationship between two adjacent tiers in an n-Tier architecture is often a **client/server** relationship. In a client/server relationship, one program (the **server**) provides services such as storing information in a database or performing computation tasks upon request from another program (the **client**). This communication is known as the **request-response**.

A tier can act as both a server and a client, fulfilling the requests of its clients and making requests of its servers at the same time.

Consider the deployment diagram below, illustrating the setup of a simple media server.



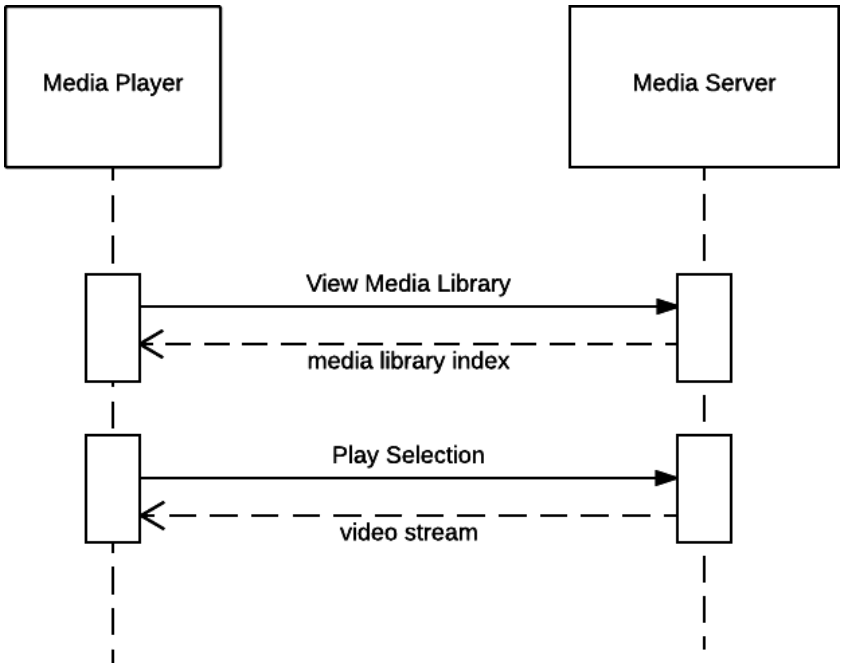
One computer hosts the media server process, where movies and TV shows are stored and can be streamed on demand. When a client requests a movie or TV show, the media server streams it to the client. The server process does not know the number of clients it may have. More clients can be added as necessary.

In this example, server and client processes are on different computers; however, they can run on the same machine. The media server can be hosted on computer and use either a media player on the same desktop or a mobile device such as a tablet to watch the media.

Client-host and **server-host** refer to machines that host client software and server software respectively. In this diagram, the top row illustrates client-hosts, and the bottom row illustrates server-hosts. This diagram is therefore a two-tier architecture. The client-hosts make up the client tier, and the media server is the data tier.

> ***Instructor's Note:***
>
> *A server is usually specified by the type of service it provides. For example, these servers might be called web servers, application servers, file servers, and print servers.*

The request-response relationship from this example can also be illustrated in a sequence diagram.



The client, in this case, a media player, requests to see what the media server has in its library. The server sends that information back. After the user makes a selection, the client requests the chosen video from the media server, which starts streaming it.

Request-response relationships can be **synchronous** or **asynchronous**.

- A **synchronous** request-response relationship is one where the client sends a request, then awaits the server's response before continuing execution. In UML, synchronous messages are depicted with a closed arrowhead. Servers do not always respond quickly, so a synchronous message may cause the client user interface to freeze while waiting for a response.
- An **asynchronous** request-response relationship is one where the client sends a request, but control returns right away, so it can continue its processing on another need. None of this processing can depend on the response from the server. Once the server has completed the request, a message is sent to the client, which will have a handler to process the response. In UML, an asynchronous message is depicted with a line-arrowhead.

Client requests are often designed to run asynchronously, because it is often preferable to have clients be responsive for other tasks even if they are waiting for information from the server.

Limiting client/server relationships to request/response messaging patterns allows for systems that can be scaled more easily by adding clients. Clients and servers are extensible, because as long as a server receives a message it knows, it will respond. The source of the message is not important to the server, so many clients can be added as needed.

n-Tier architecture can take on different numbers of tiers, based on separation of concerns. Additional tiers can be added to the system for specific purposes. The example above, of a simple client-server relationship with one server and one set of clients, is known as a two-tier architecture.

However, you could also build a three-tier architecture, where a tier is inserted between the database and end users. This tier is often referred to as a middle layer, a business layer, or an application layer—its name depends on its main responsibility. The middle layer will be a client of the database, and a server for the client application software on the end users' devices. Middle tiers help determine how or when data can be changed, and in what ways. Having a middle layer allows client application software to be thinner, as it can be focused on presentation only. This makes it easier to maintain.

You could add even more tiers to your architecture if you want, as long as the tiers serve a specific relationship.

Two potential drawbacks of n-Tier architecture include:

- It requires extra resources to manage the client/server relationships. If more tiers are added, there are more machines or processes to manage, with different communication protocols. This makes a system more complex, and therefore more difficult to change and maintain.

- A server acts as a central point of failure. Systems may have backups or mirrors, but it can take time to switch to these backups and recover the server. Redundant servers are possible but add complexity.

The advantages of n-Tier architecture are notable, however, and contribute to its common use in the software industry.

- n-Tier architecture is very scalable. Clients can continue to be added as long as a server can handle all the requests it receives in a reasonable time.
- Centralization of functionality allow for data to reside on one machine but to be accessible by any machine on the same network.
- Centralization of computing power allows client machines to require less processing power. Companies can thus offer processing power as a service, which is more practical and cost effective.
- n-Tier architecture supports separation of concerns. Middle layers can take the role of managing application logic and accessing the database directly. Adding tiers can further allow for loose coupling and levels of abstraction, making a system that is easier to change and extend.

When a system can be split into service roles and request roles, a client/server architecture should be considered.

Interpreter-Based Systems

In some software systems, users can write scripts, macros, or rules that access and compose the basic features of the system in new ways.

For example, users can write formulas in Microsoft Excel. A component known as an **interpreter** is built into the system that allows Excel to carry out the calculation in the formula "at run time". The interpreter "interprets" the formula into an intermediate representation and "executes" the representation. The end user does not need to know about the underlying implementation of the function. Systems use interpreters to drive programmable actions specified by the user.

Interpreter-based systems allow end users to write scripts, macros, or rules that access, compose, and run the basic features of those systems in new and dynamic ways. It provides users with flexible and portable functionality that can be applicable in a variety of commercial systems.

Interpreters can run scripts and macros. Scripts are often used for automating common tasks, such as scheduling tasks, performing repetitive actions, and composing complex tasks that invoke other commands. Macros are an evolution of scripts that became popular with the introduction of graphical user interfaces. A macro records keyboard and mouse inputs, so they may be executed later.

With an interpreter, end users can add functionality to a system and extend existing functionality of a system, by composing pre-existing functions together in a specific sequence, in order to create something new. These pre-existing functions are defined by the system architecture and offered to the user. The developer does not need to implement all possible combinations of functionality.

Interpreter-based systems encourage end users to implement their own customizations. This is further supported by systems that offer an easier-to-use language with domain-specific abstractions suited to the needs and thinking of the end users. End users do not have to know the programming languages used in developing the software in order to customize the system for their needs.

Interpreter-based systems offer certain advantages. In particular, interpreters make systems more **portable**, so they can work on platforms that the interpreter supports. This is an important feature with the growth of virtual machines and virtual environments—more and more services are being hosted in the cloud.

However, interpreter-based systems also offer certain disadvantages. Interpreters can be slow. Basic implementations spend little time analyzing the source code and use a line-by-line translate and execute strategy. This is a trade-off: it may be faster and more flexible for developers and end users to use an interpreted language, but slower for computers to execute interpreted code.

DID YOU KNOW?

Java also uses interpreters!

In Java, programs are first translated into an intermediate language that is loaded into a Java Virtual Machine (JVM), which executes the intermediate language. The JVM will attempt to optimize the intermediate instructions by monitoring the frequency at which instructions are executed. The instructions that are executed frequently get translated into machine code and executed immediately.

On the next execution of the same intermediate instructions, the JVM uses lazy linking to point the program to the previous machine code translation. Instructions that are not used frequently are left for the interpreter of the JVM to execute.

This decreases execution time since frequently used instructions do not need to be constantly translated and the entire program does not need to be translated all at once. The JVM also provides portability to Java programs, allowing them to run on many operating environments.

Dataflow Systems

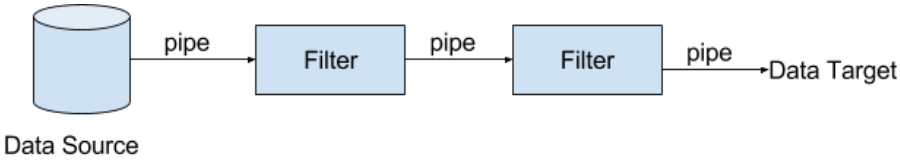
Data flow systems involve a series of transformation on sets of data. Data is transformed from one form into another using different types of operations.

This section will explore designing a data flow system through the **pipe and filter architectural style**, a specific type of data flow architecture.

Pipes and Filters

A pipe and filter architectural style has independent entities called **filters**, which perform transformations on data input they receive, and **pipes**, which serve as connectors for the stream of data being transformed.

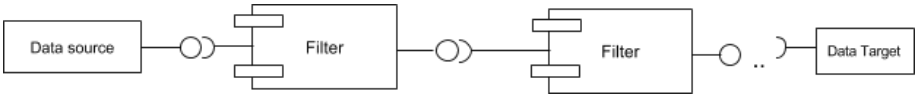
Below is a simple diagram outlining how a pipe and filter architecture works.



Data flows in one direction, beginning at the data source and moving through a series of pipes and filters, to end at the data target. Changes to data are done sequentially from filter to filter. Each pipe serves as a connector for the stream of data. It transmits the output of one filter as input for the next filter. Each filter performs its local transformation on the data input it receives from the previous pipe and then sends the output on to the next pipe.

The order in which the filters transform data may change the end result. This is similar to mathematics—the order in which addition or multiplication occurs in an expression can change the end result.

UML component diagrams are a good option for showing the details of pipe and filter architecture.



The data source can be connected to a number of different filters. Each filter has a required interface to receive input from data sources and other filters. They also have a provided interface that will output transformed data. The data target is connected to receive the fully transformed result.

Advantages to using the pipe and filter architecture include:

- Loose and flexible coupling of components. In particular, this applies to filters. Each filter runs independently of other filters, which allows the easy addition of new filters, of moving filters around, or of changing individual filters in a system. ‘

- Filters can be treated as black boxes. Users of the system do not need to know the inner workings of each filter.

- Reusability. Each filter can be called and used over and over again with different inputs. Filters can be repeatedly used in many different applications for different purposes.

Disadvantages to using the pipe and filter architecture include:

- Reduced performance due to excessive overheads in filters. Each filter must receive input, parse that input into some data structure, perform transformations, and send data out. If every filter must do this, it is a lot of overhead, and the performance of the system will be affected as more and more filters are added.

- Filters may get overloaded with massive amounts of data to process. If one filter is working to process large amounts of data, other filters may be waiting for inputs.

- This architecture cannot be used for interactive applications. Applications that require rapid responses will not find this architecture suitable, as data transfers and transformations take time depending on the amount of data being transmitted.

Pipe and filter architectures are popular in use, such as in the UNIX operating system for text-based utilities. If different data sets need to be manipulated in a system, the pipe and filter architecture is a good choice.

Implicit Invocation Systems

In **implicit invocation systems**, functions are not in direct communication with each other. The **event-based architectural style** is one design for such a system. This style derives from the event-driven programming paradigm.

Event Based

In event-based architectural styles, the fundamental elements in the system are **events**. Events are both indicators of change in the system and triggers to functions. Events can be signals, user inputs, messages, or data from other functions or programs.

Under the event-driven programming paradigm, functions take the form of event generators and even consumers. Event generators send events, and event consumers receive and process these events. A function can be both an event generator and an event consumer.

In the event-based architectural style, functions are not explicitly called. Instead, event consumers are based on events sent from event generators. Event-based functions experience **implicit invocation**, where communication between functions is mediated by an **event bus**. An event bus is the connector between all event generators and consumers in the system.

To achieve this structure, an event and an event consumer must be bound via an event bus. This means that each event consumer registers with the event bus to be notified of certain events.

When the event bus detects an event, it distributes the event to all appropriate event consumers. The event bus is a critical part of the system. The observer design pattern actually manifests the event-based architectural style.

One means of implementing the event bus is to structure the system to have a main loop that continually listens for events. When an event is detected, the loop calls all the functions bound to that event. This function must likely also be an event generator, because often an event consumer will have to notify other functions when it has completed its task or to send a state change. This function will implicitly invoke other functions to run after it has completed. It does this by sending out an event to the event bus. When an event reaches the event bus, corresponding event consumers will be triggered and the next computation can take place.

When designing your system, it is important to take into account that indirect communication between functions may not be as efficient as explicit function invocation. In event- based architectural styles, event generators do not necessarily know which functions will be consuming their events, and likewise, event consumers do not necessarily know who is generating the events they handle. This is called loose coupling, which makes it easier for systems to evolve and scale up—adding a new function requires only registering a new event-function pair to the event bus and adding a new event consumer.

Updating objects and data in an event-based system must also be considered.

Event consumers can change shared states as they process an event if objects in the system are globally accessible. However, this allows event consumers can be called asynchronously, that is, an event consumer does not need to wait for other event consumers to finish running before running itself. Two event consumers could be running at the same time on shared data.

This asynchronous design may increase efficiency of the system, but it can also result in race conditions. Race conditions mean that the correct behaviours of functions is sensitive to timing or order, so shared data may not be updated correctly.

Functions can be coordinated for access to shared data through a **semaphore**. A semaphore is a variable or abstract data type that toggles between two values, available and unavailable, and that is used to control access to shared data. Available indicates that the shared data is not in use, and unavailable indicates that the shared data is in use by a function.

In general, a semaphore has a special operation to check and toggle its value in one step. Any function that would like to access the shared data must first check the value of the semaphore. If the value is set to unavailable, then the function must wait. If the value is set to available, the function can access the shared data, and the value is set to unavailable. When the function no longer needs access, it will toggle the semaphore value to available.

If an object is not globally accessible and its state is changed by an event consumer, then some indication must be sent back to the event bus before the function ends. Event consumers can thus be aware of the state change to that event.

In event-based architectural styles, events and functions do not behave in a predictable way. There is no guarantee of exactly when an event will be handled, how long it will take to handle, or when an event generator will emit an event. Control flow is thus based on which events occur during its execution and in what order.

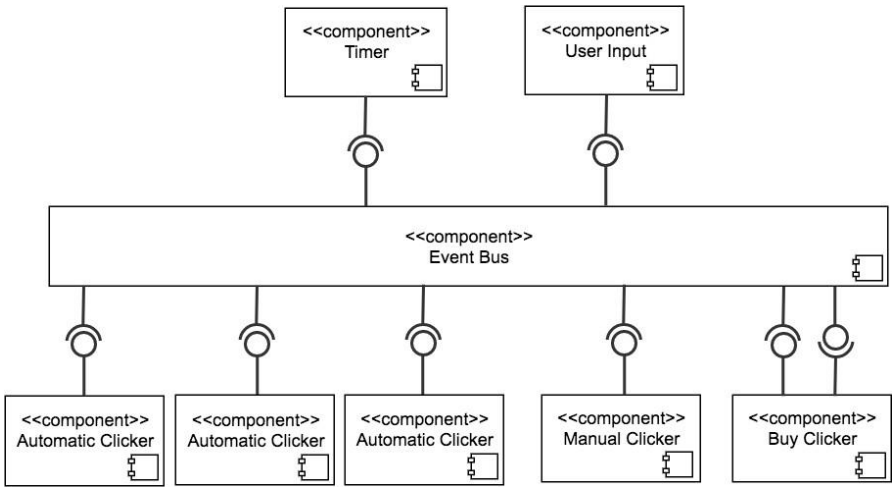
Event-based architecture styles are well suited to interactive applications. Graphical user interface applications that rely on user input and distributed systems that interact with other programs are prime candidates for this style.

Cookie Clicker Example

Let us examine event-based architecture through an interactive game, “Cookie Clicker.” The goal of this game is to collect as many points as possible by clicking on an image of a cookie with your cursor. This can be done manually or by clicking on a cursor icon.

This icon has a functionality—it will purchase a special automatic clicker that automatically clicks the cookie at regular time intervals, thus reducing work on the user's part to collect points. Users can "purchase" multiple cursors at the cost of some points.

Examine the component diagram below to see how this can be designed with an event-based architecture.



A main event loop listens for event. Depending on where users click within the window, a different event consumer will be invoked. When the cookie is clicked, the function registered to handle this click event is called and total points increases by one. When the "buy clicker" icon is clicked, then the function registered to handle this click event is called. This first reduces the score, as a purchase was made. Then, an automatic clicker function is added to the system, along with a timer function.

The timer function is an event generator, registered to the event bus, that sends a "timer event" every five seconds. The event bus detects when such an event is emitted and triggers every automatic clicker function to consume this event. Each automatic clicker function is responsible for making one click to the cookie. Every time a timer event is sent to the event bus and received by the automatic clicker functions, the total points increase by the number of automatic clicker functions. These event consumers, event generators, and event bus form the system of the game.

Process Control Systems

Process control is important in many processes to ensure efficient and safe operations.

Feedback Loops

A **feedback loop** is one of the most basic forms of process control. It has four basic components: a sensor, a controller, an actuator, and the process itself. The sensor monitors some kind of important information. The controller is the logic of the system. The actuator is the physical method of manipulating the process. The process is what is trying to be controlled.

For example, consider controlling room temperature. This is a feedback loop. A thermostat acts as a sensor for the room's temperature. The software is the controller. The heating vent in the room is the physical means to manipulating the process. The room temperature is the process.

In a feedback look, the controller logic must run continuously. This is because the process is always changing.

The frequency, or how often the loop runs, depends on the desired level of control and the sensitivity of the system. For example, room temperature changes slower compared to other processes, so it is not necessary or useful to have a high frequency, such as every microsecond.

Feedback loops can often be improved. Consider the example above of room temperature. Let us improve that loop by introducing a proportional controller. A proportional controller calculates an error value as the difference between the desired setpoint and a measured process variable and then applies a correction based on proportion. In this case, a heating vent whose position can be controlled could read an error, and open the vent according to how big the error signal is. As the temperature comes closer to the setpoint, the heating will slow down so that the setpoint can be approached gradually. (Proportional controllers can also have undesirable effects, but this runs outside the scope of this course.)

Other Variations

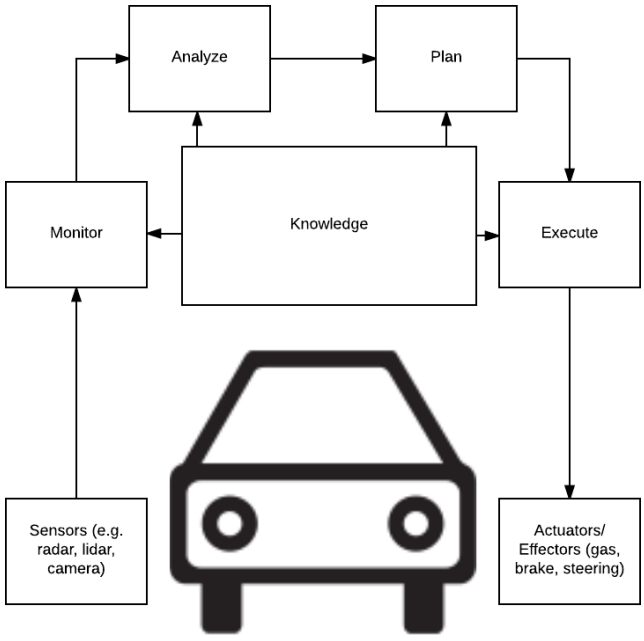
Other variations exist on process control.

A feedback loop is an example of a closed loop, as information from the process is used to control the same process. An open loop is one where the process is controlled without monitoring the process. Open loops cannot deal with changes in the process or check itself to see if it is succeeding.

Feedforward control occurs in systems with processes in series. Information from an upstream process can be used to control a downstream process. Feedforward control requires a good model of how process response and deals with unknown events. These systems are often used in conjunction with feedback loops.

For example, a flood protection system could use feedforward control. If an upstream monitoring station detects large flow rates, then the controller for the flood protection system could be signalled. The controller asks the actuator, a gate, to open and divert water into a reservoir to keep water levels manageable.

More complex problems will have many sensors and ways to control the process. Process control architecture for complex problems will therefore be more complex. Below is a diagram for a complex example of a self-driving car.



This architecture is based on a MAPE-K structure, which has four major steps: monitor, analyze, plan, and execute. All four of these steps must have knowledge of the process.

Step	Description
Monitor	The component or collection of components that interfaces with sensors, turning raw signals into meaningful information for the software.
Analyze	The software must next analyze this data. Data could be integrated from several sensors, so analysis might be quite sophisticated. More sophistication generally requires a better model of the process.
Plan	Next, planning needs to be done. More advanced planning needs will require more knowledge. This knowledge is in the form of a model of the system. Such models should be designed in close conjunction with engineers specializing in the process.
Execute	The final step is to execute. Analysis and planning should make execution relatively simple. They will simply turn desired actuator positions into signals to communicate to those actuators.

Knowledge is very important in MAPE-K systems, particularly complex ones. Currently, technology companies are investing heavily in such systems, due to the advent of machine learning and big data. This investment goes towards real-world testing, machine learning, and data modelling in order to operate these systems. Effective software architecture is key. Even though it is handled by specialized professionals, it is important to be aware of process control. Software is becoming a bigger part of controlling physical systems.

MODULE 3: ARCHITECTURE IN PRACTICE

Upon completion of this module, you will be able to:

(a) Analyse, evaluate, and describe how software architectures interact with software development.
(b) Identify what makes an architecture good or bad for a given problem.
(c) Evaluate software architecture quality, and identify improvements.
(d) Understand how architecture can be used to plan development and structure a development team.
(e) Identify opportunities for reuse to reduce costs and increase development efficiency.

This module will focus on how software architecture is used in practice. This includes exploring how software architectures are analyzed, evaluated, and how they interact with software development. This module also explores what makes an architecture good or bad for a given problem, how architecture is evaluated for quality, how to identify improvements in software architecture, how to use architecture to plan development and structure a development team, and how to identify opportunities for reuse of architecture to reduce costs and increase development efficiency.

Quality Attributes

Software architecture aims to combine software design patterns and principles in order to define the software's elements, their properties, and how the elements interact with each other. This suggests that these elements alone can determine the capabilities and the quality of software architecture. However, as design patterns address only a specific problem and not all business needs, this is not the case. Modern systems need to be able to focus on a wide range of problems, not just technical issues.

Individual design patterns and principles are concerned primarily with functional software issues, while software architecture considers functional and non-functional requirements. Software architecture must address the big picture. Architecture sets guidelines for design patterns and principles in order to ensure **conceptual integrity** and consistency throughout.

For example, design patterns covered in the previous course do not touch a number of software qualities, such as testability, usability, and availability. Software architecture addresses these qualities by carefully structuring and coordinating design patterns in order to construct a unified system. Software architecture is qualified by how well the “entire” combination of software elements work together to provide functionality and handle potential system issues as well as how well the design addresses user experience and ease of development.

Software architecture does not have inherent qualities that make it “good architecture” or “bad architecture.” Instead, software architecture is designed to address a set of requirements. Requirements are used to address a problem or need. Further, different architectures are meant to be used in different environments and contexts. This means that an architecture isn’t bad in some cases, but rather just be incorrect for the context it is being used in.

Most systems use a combination of architectural designs. The requirements for modern systems are complex, and there may not be a clear-cut software architecture that can address all of them. This is particularly true for non-functional requirements, which are not always clear or explicitly presented by clients and stakeholders. Non-functional requirements can also vary between different stakeholder groups. For example, an end user may not care about testability, while a development group will.

Software architecture should address the concerns of all stakeholders, however. Prioritize system requirements and don’t be afraid to make compromises in order to find a good balance between system qualities.

If system architectures are not measured as inherently good or bad, there must be a way to determine if the architectural design is capable of meeting system requirements. This quality of a system is determined by quality attributes. Quality attributes are measurable properties of a system used to gauge a system's design, run-time performance, and usability.

For an attribute to be measurable, there needs to be an objective means of quantifying it. For example, system availability can be measured by the system's uptime in some unit of time.

This section presents industry consensus for quality attributes. However, these can be added or removed from your evaluation of a system, based on specific needs. As always, context is important in evaluating what is best for your system architecture.

This specialization has emphasized **maintainability**, **reusability**, and **flexibility** when designing and implementing systems. These attributes will be covered, as well as other important attributes from a developer perspective, as outlined in the table on the following page.

Quality Attribute	Measurement of
Maintainability	The ease at which your system is capable of undergoing changes. Systems will undergo changes throughout its life cycle, so a system should be able to accommodate these changes with ease.
Reusability	The extent in which functions or parts of your system can be used in another. Reusability helps reduce the cost of re-implementing something that has already been done.
Flexibility	How well a system can adapt to requirements change. A highly flexible system can adapt to future requirements changes in a timely and cost-efficient manner.
Modifiability	The ability of a system to cope with changes, incorporate new, or remove existing functionality. This is the most expensive design quality to achieve, so cost of implementing changes must be balanced with this attribute.
Testability	How easy it is to demonstrate errors through executable tests. Systems should be tested as tests can be done quickly, easily, and do not require a user interface. This will help identify faults so that they might be fixed before system release.
Conceptual Integrity	The consistency across the entire system, such as following naming and coding conventions.

In addition to qualities to account for from a developer's perspective, it is also necessary to take into consideration qualities from a user's perspective. The table below covers some of these quality attributes and what they are measuring.

Quality Attribute	Measurement of
Availability	The amount of time the system is operational over a set period of time. A system's availability is measured by uptime, so it can be determined how well the system recovers from issues such as system errors, high loads, or updates. A system should be able to prevent these issues from causing downtime.
Interoperability	The ability of your system to understand communications and share data with external systems. This includes the system's ability to understand interfaces and use them to exchange information under specific conditions with those external systems. This includes communication protocols, data formats, and with whom the system can exchange information. Most modern systems are interoperable and do not exist in isolation.
Security	The system's ability to protect sensitive data from unauthorized and unauthenticated use. This attribute is important as most systems generally contain sensitive information. This information should be protected from those not authorized to see it, but readily available to those who have authorization. Closely associated with this is data integrity. A system should be able to control who can see the data versus who can change it.

Performance	The system's throughput and latency in response to user commands and events. Performance determines how well your system is able to respond to a user command or system event. Throughput is the amount of output produced over a period of time. Latency measures the time it takes to produce an output after receiving an input. This attribute has a major influence on architecture due to its importance to users. It is better to have a lower price per unit of performance. With the advancements in technology, this ratio has been steadily decreasing.
Usability	The ease at which the system's functions can be learned and used by the end users. Usability determines how well your system is able to address the requirements of end users. In order to have high usability, your system needs to be easy and intuitive to learn, minimize user errors, provide feedback to the user to indicate that the system has registered their actions, and make it easy for the user to complete their task.

As software is not inherently good or bad, it is very important to create or choose an appropriate architectural design for your systems. The system architecture will ultimately determine how functionality is implemented, how subsystems communicate with each other, and how end users will interact with the system.

Software architecture is also important, as it makes maintaining, supporting, and updating the system throughout its life cycle much easier. If the qualities of a system are poor, then it will be difficult to support changing needs.

A high-quality system does not need to be “complex.” An overly complex system makes it difficult and time consuming to produce. It is good practice to try to minimize complexity in the design. If a simpler architecture can satisfy all the system requirements and achieve a high-quality design while needing less time and less money, it makes sense to go with that.

It is important to have detailed and up-to-date documentation of a system. In fact, this is part of a high-quality system. This helps record and share an architectural vision. The documentation keeps a design cohesive, should the architect or any of the developers leave the team.

Creating a high-quality system architecture should be methodical and use a set of rules or guidelines for the design process. Although every company will have some specific design rules or system structure rules, some should be common sense.

Design process guidelines include:

- Recognizing the important of quality attributes and prioritizing them for each system being designed. Quality attributes should be kept up to date throughout the life cycle of the system. Trade-offs and details of how a system will meet qualities should be noted.

- Involving a technical lead in the design process. Although architectural design can be applied to many different technologies, involving a technical lead will help identify any implementations that may pose a challenge, which may need to be re-considered in the design.

- Taking a design approach from the perspective of different groups of stakeholders.

Structural rules that ensure there is conceptual integrity when implementing the system include:

- Having well-designed subsystems that are assigned responsibilities based on design principles. Remember that subsystems should use design principles such as separation of concerns.

- Having consistent implementation of functions across the entire system. If the same task needs to be achieved by two or more subsystems, then consider how implementation can be reused.

- Having a set of rules on how resources are used. Resources may include memory, bandwidth, or threads that the system has access to.

In summary, when selecting the appropriate architecture for the problem at hand, it is important to involve all groups of stakeholders in the design of the system, to adopt good documentation practices, and to set rules for design and implementation.

A well-designed system considers quality attributes from a developer's perspective, which includes maintainability, reusability, flexibility, modifiability, testability, and conceptual integrity; the system should also consider attributes from a user's perspective, which includes availability, interoperability, security, performance, and usability.

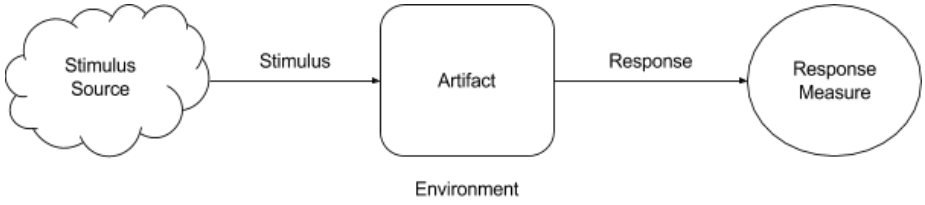
Analyzing and Evaluating an Architecture

In addition to designing a system, it is important to know how to evaluate the design to determine if it addresses the concerns, or the requirements, of all stakeholders. Analyzing and evaluating software architecture can be difficult due to the abstract nature of software.

It is important to methodically analyze and evaluate a system's behaviours, quality attributes, and various characteristics. This section will look at how individual qualities are specified for evaluation, and how to analyze the architecture as a whole.

In order to measure quality attributes, **quality attribute scenarios** can be used to determine if a system is able to meet requirements set for the attribute. There are two kinds of scenarios:

- A **general scenario**, which is used to characterize any system
- A **concrete scenario**, which is used to characterize a specific system



Both general and concrete scenarios have:

- A stimulus source, which is anything that creates a stimulus. It can be internal or external to the system.
- A stimulus is a condition that will cause the system to respond. As the source of the condition can originate internally or externally, types of conditions will need to be differentiated.
- An environment is the mode of the system when it is receiving a stimulus. This is an important aspect, particularly if the system involves distributed computing, or if it can exist in operational modes besides just running and stopped, like recovering from a failure.
- An artifact is the part of the system that is affected by the stimulus. In large-scale systems, a stimulus should not directly affect the entire system.
- A response is how the artifact will behave as a result of receiving a stimulus. This response could include handling an error, recovering from a failure, updating system logs, dispatching security alerts, or changing the current environments.
- A response measure is a metric used to quantify the response so that the quality attribute can be measured. The metric should be quantitative and objective, such as probability of failure, response time, repair time, and average system load.

Scenarios are built to identify situations that impact the quality attributes of a system. In the context of analyzing and evaluating architecture, you should focus on situations that are outside of the normal execution path. This means that scenarios involving incorrect input, heavy system loads, or potential security breaches should be prioritized highly.

It is a system's "inability" to handle unexpected failures that stops it from achieving a specific quality attribute.

Availability Example

Let us consider scenarios through an example. Imagine you are addressing the availability of a system. In addition to focusing on when a system is online and behaving normally, you have to consider situations where the system becomes unavailable.

In a general scenario, high-level events are summarized. As a single scenario may involve many component values, it often summarized in a table.

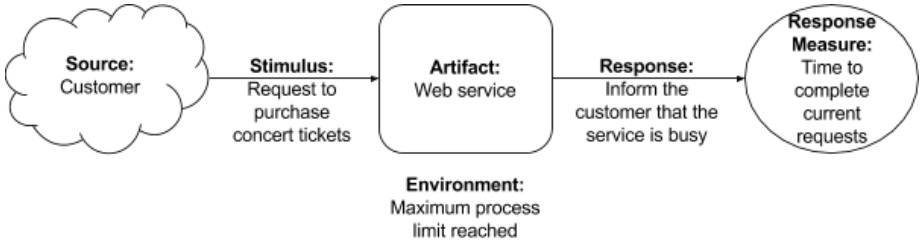
Below is an example of a general availability scenario.

Scenario Component	Scenario Component Value
Stimulus Source	- End user - Internal subsystems - External subsystems
Stimulus	- Incorrect user input - Internal exceptions - Unrecognized system request - High request volume - Heavy system load
Environment	- Normal operating environment - Starting up - Shutting down - Heavy load operating environment - Recovering from error - Processing request
Artifact	- System servers - System process
Response	- Log exceptions - Notify user - Send error response to external system - Redistribute data processing - Redistribute system requests - Notify user and external systems that the system is shutting down/starting up
Response Measure	- Time to restart (shutdown and startup sequence) - Time to undergo recovery - Time to complete requests when the request volume is high - Time to complete a process under heavy system load - Time to become available after encountering an incorrect input or request

Concrete scenarios are more focused. They can help you test an architecture with a specific stimulus under specific system environments and measure how well the system can respond.

Let us expand our example on availability. Availability of a web server can be hindered in its ability to process requests when at resource limits or under heavy load. A server's availability should be measured under different conditions.

Consider the specific concrete scenario below, where a customer needs to wait for the system to finish processing previous orders for concert tickets before they are able to send their purchase request. The system is unavailable for customers if the system can't accept their request for tickets, which has a negative effect on the availability quality attribute of the system.



Architecture Trade-off Analysis Method

There are a number of methods that can be used to implement scenarios in analysis and evaluation of the entire architecture. This section will focus on an evaluation technique developed by the Software Engineering Institution at Carnegie Mellon University known as Architecture Trade-off Analysis Method (**ATAM**).

With ATAM, evaluators do not need to be familiar with the architecture or the problem space. A system can be evaluated by outsiders.

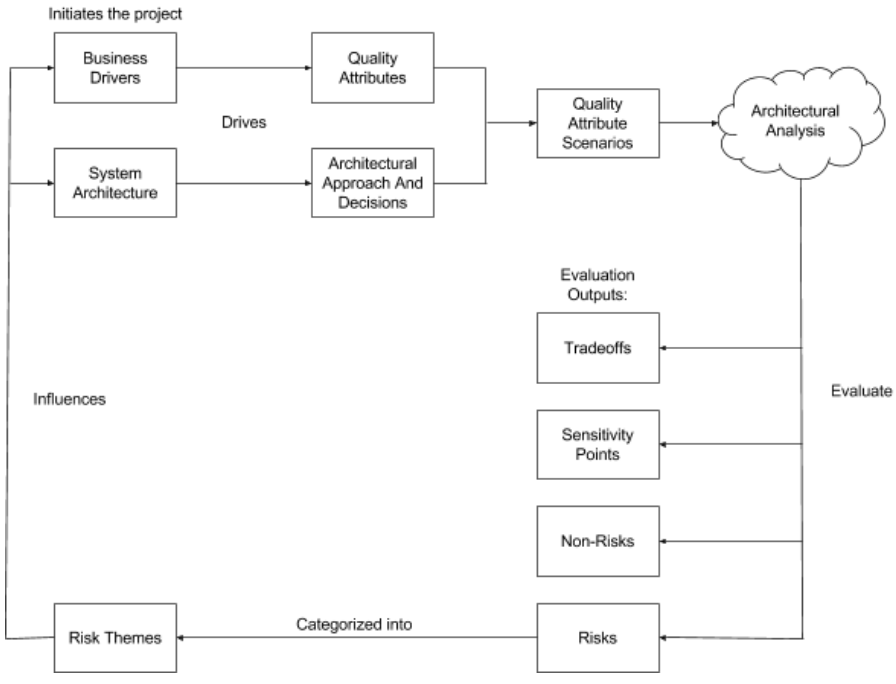
There are three different groups of participants in ATAM.

- The first is the **evaluation team**. This has three different subgroups: designers, peers, and outsiders. Evaluation teams must be unbiased.
    - **Designers** are the subgroup involved with architectural design. They naturally follow iterative, hypothesis-driven methods when designing, including analyzing systems requirements, creating a design to address those requirements, and reviewing the design.
    - **Peers** are the subgroup composed of those who are part of the project, but they are not involved in the design process. Their point-of-views helps round out design decision.
    - **Outsiders** are the subgroup composed of those external to the project or organization. Involvement of outsiders helps eliminate bias towards the project in evaluation. Outsiders should have experience and expertise in analyzing architecture.
- **Project decision makers** are a participant group of project representatives with the authority to make project decisions. This could include project managers, clients, product owners, software architects, and technical leads.
- **Architecture stakeholders** are a participant group of those who want the architecture to successfully address business needs, but who are not actively involved in the evaluation process. This could include end users, developers, and support staff.

In an ATAM, a software project is initiated when business drivers identify a need for a system to address some problem. Business drivers go hand in hand with the system architecture, which is created as the solution to the business issues. Together, business drivers and system architecture determine the quality attributes of the system, the architectural approach taken, and the design decisions that are made. These combine together to create quality attribute scenarios.

Scenarios can then be analyzed, resulting in an evaluation of the system, which includes trade-offs, sensitivity points, non-risk scenarios, and risk scenarios. Since the risk scenarios have a negative impact on the quality of the system, each of them are analyzed and categorized into “risk themes.”

The diagram below illustrates this high-level process for ATAM.



The entire ATAM process itself can be broken down into nine steps.

1. **Present the ATAM**.
   The evaluation team presents the ATAM process. This includes: the context for the evaluation, expectations, procedures, outputs, and addresses any concerns about the evaluation.

2. **Present the business drivers**.
   The project decision makers present the business problem and the goals for the system as well as the system’s features and requirements, project constraints, and scope.

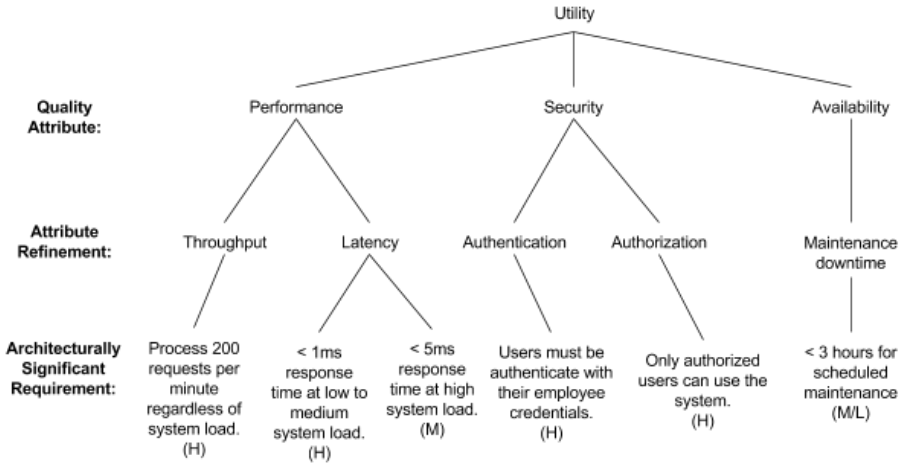
3. **Present the architecture**.
   Both current and expected state of the architecture is presented as well as constraints such as time, cost, difficulty of the problem, and quality expectations.

4. **Identify the architectural approaches**.
   This analysis activity involves examining the architectural patterns that have been used in the system. This step analyzes the documentation and the notes from presentations and asks questions to gain more clarity about the system.

5. **Create a quality attribute tree**.
   A quality attribute utility tree is created, which maps the quality-related **architecturally significant requirements** (ASR)s for each quality attribute. ASRs arise from the business drivers.

   To build such a tree, the overall “utility” of a system is broken down into quality attributes, which are refined into attribute refinements. **Attribute refinements** are more specific qualities of a system. Once the quality attributes have been refined, ASRs can be associated with the appropriate attribute.

   This step provides insight into the system and identifies quality priorities. This step should be conducted in conjunction with project decision makers.



In quality attribute trees, ASRs are given priority values to denote if they are “must-haves” or not. The example above uses high (H), medium (M), and low (L) designations, but these values could differ from system to system.

6. **Analyze the architectural approaches**.
   Using the prioritized ASRs from the utility tree, examine the architecture and determine how it addresses each ASR. This allows for the identification and documentation of risk and non-risk scenarios, **sensitivity points**, and **trade-offs**. This step reveals the capabilities of the system and consequences of design decisions. It is not meant to be comprehensive, but it identifies connections between business drivers and system architectures.

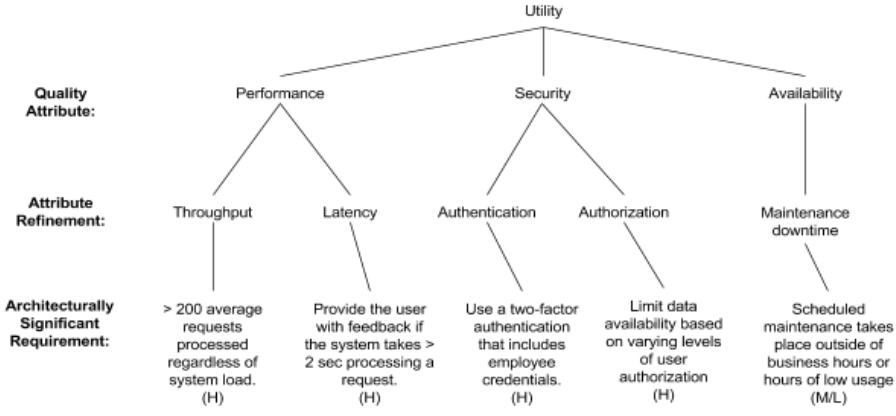
7. **Brainstorm and prioritize scenarios**.
   In this step, each group of participants creates quality attribute scenarios that are important to them, which they would expect when using the system. Scenarios with similar quality concerns or behaviours can be merged.

   This step provides an overview of the day-to-day system usage and insights into the environment that the system can be in.

Scenarios are prioritized based on importance to each stakeholder, and the evaluation team compares the list with the prioritized ASRs in the utility tree. If the priorities of the stakeholders match closely with the priorities in the utility tree, then there is **good alignment**.

There is a risk if there are many additional scenarios discovered during this step that were not in the original set of ASRs. It is an indication that the architecture is not able to address the needs of stakeholders.

8. **Re-analyze the architectural approaches**.
   Recreate a utility tree, but this time, using the top five to ten scenarios prioritized in the previous step. This new tree can be used to talk with the system architect and discover how each scenario can be achieved with the system design.



9. **Present the results**.
   Results of the evaluation are compiled together. This includes all architecture documents, utility trees, risk and non-risk scenarios, sensitivity points, trade-offs, and risk themes. Risk scenarios should be grouped together by risk theme. Risk themes help identify which business drivers are affected.

ATAM helps expose unseen risks for stakeholders involved in the architectural process.

Modern systems are becoming more and more complex, and creating an architecture that can achieve all the requirements for quality attributes is becoming increasingly important. Being able to evaluate and analyze architectures helps successfully create high-quality systems.

ATAM is a common method for analyzing and evaluating architectures, especially as it does not require evaluators to have intimate knowledge of the system, and covers the viewpoints of all important stakeholders. ATAM helps minimize risks in a system by identifying them and helps architects minimize the effects of sensitivity points and be sensible about trade-offs.

ATAM also helps facilitate communication between stakeholders, including identifying issues with newly discovered functionalities that the stakeholders expressed to be important.

Relationship to Organizational Structure

A study from the Harvard Business School showed that looser organizational structures, such as in open-source projects, led to more loosely coupled modular code (MacCormack, Rusnak, & Baldwin, 2007). On the other hand, in-house development teams tended towards tighter coupling. They suggest that this difference could be due to conscious choices by the developers or just a natural consequence of the environment in which the software is developed.

This study supports **Conway's Law**. This law states that a software system will tend to take a form that is congruous to the organization that produced it.

This law should be taken into consideration when putting a software development team together. Rather than bringing a team together to work on a project with constant communication that can lead to a tightly coupled system, knowledge of Conway's Law should result in planning development teams around the desired architecture. Many programmers know Conway's Law explicitly or intuitively.

When putting together a team of developers, it is a good idea to first plan the architecture that works best for the system and organize the team around that architecture. For example, if you are building a web application using the n-Tier architecture, you could have one team for the data backend, one team for the application logic layer, and one for the presentation tier.

This increases the importance of flexibility in design, as there may be cases where implementation of a component is not yet known or may change.

Conway's Law may entail extra work to provide unified and scalable architectures. There may be need of a team whose purpose is to consolidate and enforce common services across the whole system.

> ***Instructor's Note***:
>
> *Even though Conway's Law is called a "law," it is actually more of an observation that can be stretched or broken in some situations. Even so, it is helpful to be aware of the law and the implications it holds for putting together a team of developers or software architecture.*

Product Lines and Product Families

As developers work, they should constantly be looking for opportunities for code re-use. Libraries, toolkits, and engines are examples of how the software industry have made use of code re-use. Code re-use, even if it requires modification, saves time and resources, and it offers proven, robust solutions. This is why well-written and well-documented code is important.

One means of harnessing code re-use is to treat a group of products as a **product line** or **product family**. These terms are somewhat interchangeable. This section will use product line.

Product lines are groupings of products that typically share the same operating system, which allow for a great deal of code re-use. A well-known example might be iOS, which is used across the product line of iPhones, iPads, and iPod touch products.

Product lines present several advantages:

- Product lines designed with similar features allow companies to save money because it reduces their development costs per product. The time saved on development can be spent testing for quality attributes, such as reliability, user experience, security, and maintainability. However, less testing is needed overall per product, because code that is reused has already been tested.
- Another advantage of product lines is user experience. If several products share characteristics, then the learning curve for customers and developers is smaller. This may also drive sales of other products in the same line.
- Product lines may also reduce time to market. Since most software components are already made, making a new product in the product line, or refining an existing one, takes less time.

If only one or two products are being produced, it might not be worthwhile to treat them as a product line, unless there is intent to develop more products later on. In product lines, there is work needed upfront to develop the foundations for products.

Implementing Product Lines

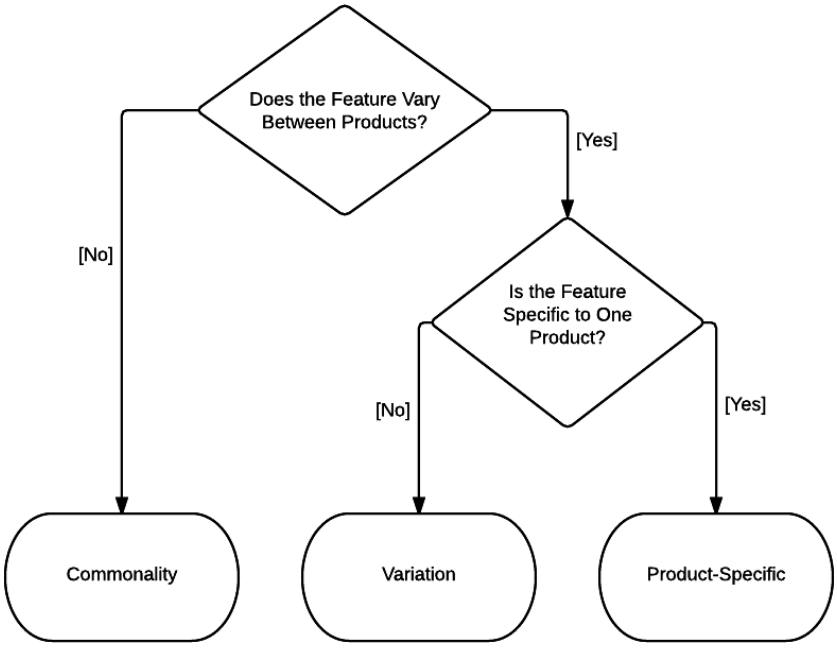
The book *Software Product Lines in Action* (2007) discusses architectural considerations for software product lines. Let us examine some of the steps outlined in this resource (see Course Resources).

1. **Commonalities**
   The features that stay the same in every product should be identified.

2. **Variations**
   The features of the product line that vary between products should be identified. For example, for a line of tablets, a variation could be support for different cellular network connections.

3. **Product specifics**
   The features that are unique to one and only product should be identified. These features will be developed by the team that is responsible for that product. For example, maybe one of the tablets developed will also specialize in reading eBooks. Its product specifics could be additional tools for managing eBooks and support for an eInk section.

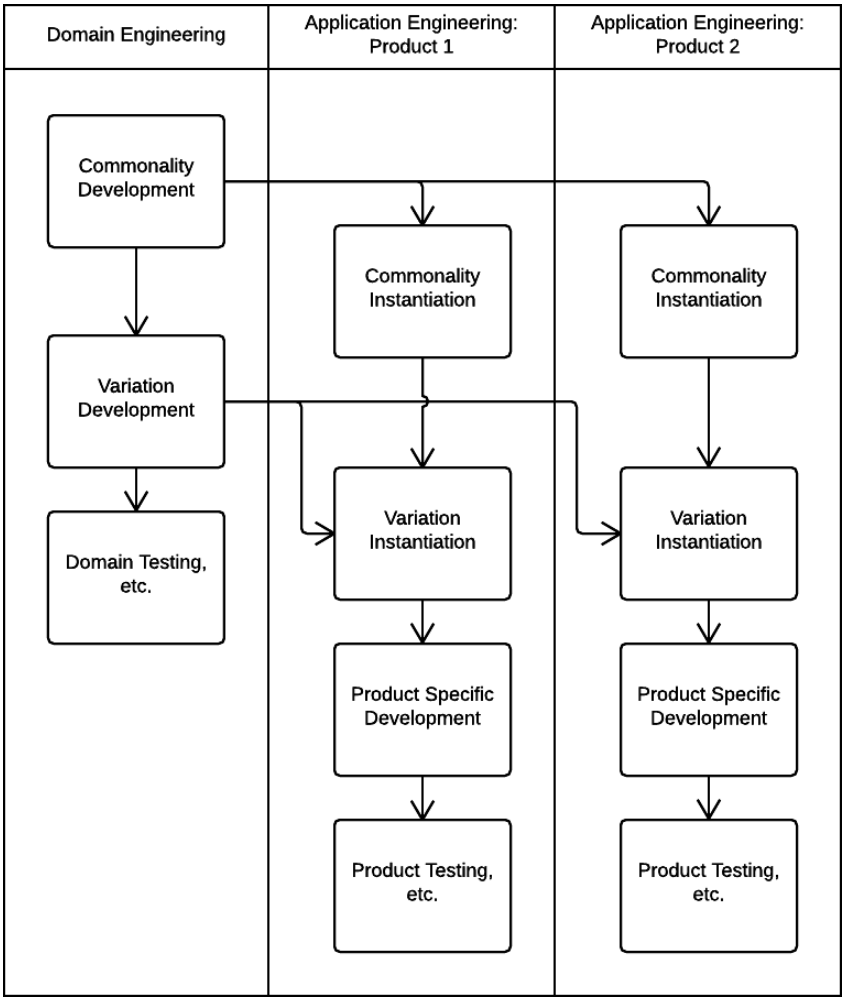
Below is a diagram of considerations for software product lines.



Once these considerations are established, you can turn to development of product lines. Product line development teams are generally divided into two camps:

- **Domain engineering –** This is the development of commonalities and variations. Essentially, domain engineering focuses on putting together the “infrastructure.”
- **Application engineering –** This is the actual development of the product. It includes using the commonalities, deciding which variations are necessary, and integrating them into the product and developing product-specific features. This is sometimes described as “instantiating” a product. There could be several application engineering teams, one for each product.

From a project management point of view, separating development into domain and application engineering allows for separate development cycles. The domain engineering team can determine the long-term needs of the product line, evolve and test components as needed, then release that infrastructure to the application team. Then, the application team can develop product-specific features using the infrastructure and the requirements of the product. They will develop product-specific features, decide what variations to use, and test the final product. This can be done while the domain engineering team is working on the next update of the infrastructure.



Reference Architecture

Products in a product line likely have similar architectures, as they serve a similar purpose. Instead of creating a new architecture from scratch for each product, the product line usually has an architecture that products can build on or change. The domain team is generally responsible for this **reference architecture**.

The reference architecture must:

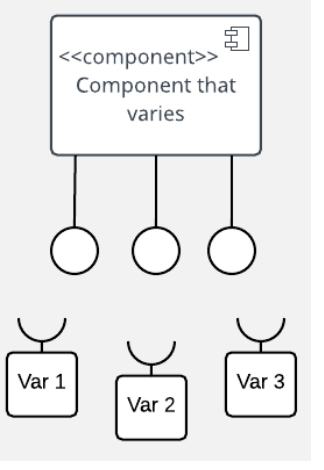
- be designed with respect to the needs of the software, while taking into account all current products in the product line

- include the capacity of variation, for differences in products and to allow for future products

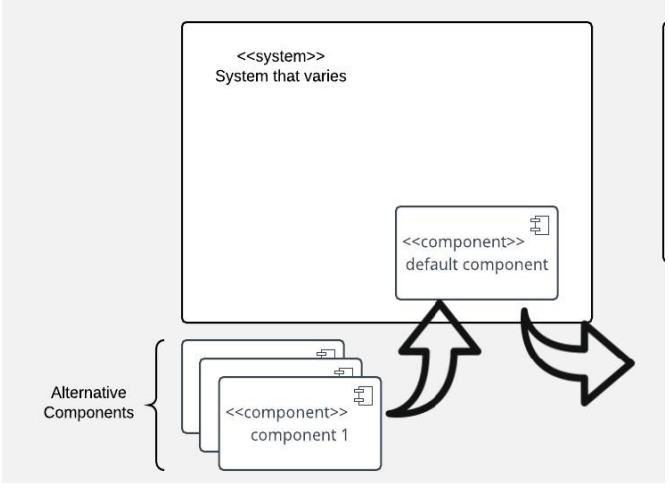
There are three general techniques to realize variations in a system:

- adaptation technique
- replacement technique
- extension technique

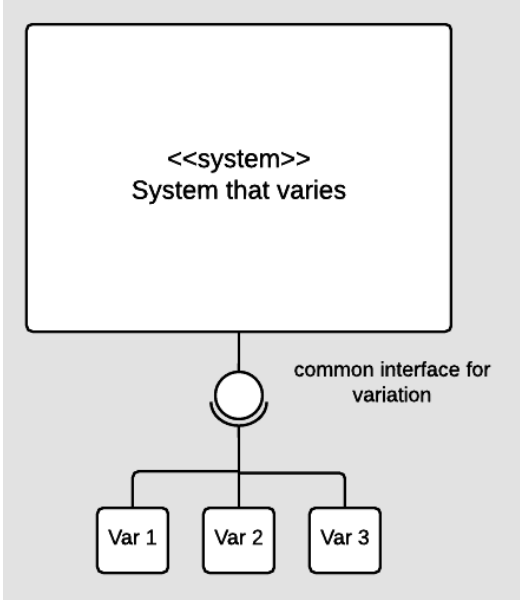
Adaptation technique. In this technique, a component has only one implementation, but it supplies interfaces to change or add on to it. Components can be adapted through settings in a configuration file, by adding new methods or by overriding existing methods. See below for a diagram of an adaptation technique.



Replacement technique. In this technique, there could be a default component that is replaced with alternatives to realize variation. There may not even be a default; instead there is a gap in the system, and the developers must supply one of the components. See below for a diagram of a replacement technique.



Extension technique. In this technique, a common interface is provided for all the variations of the system. These variations are often called extensions, add-ons, or plugins. See below for a diagram of an extension technique.



Variations can be realized at different times. Different variations may even come in at different stages—they can be formed early on, during the design or development process of the application engineering team, or they can be formed when the software is compiled and built. Similarly, variations can be realized at different times, like when the software is launched or after the software is launched, whenever they are needed.

The dynamic nature of variations puts pressure on software architecture to be highly modular. A combination of the nature of the variation, the type of reference architecture, and the software requirements determines what variation techniques are available to architects, and how they are implemented. As with any design decision, there are trade-offs to consider when building variation into architecture.

This concludes the Software Architecture course notes. The next course in this specialization will be focused on Service-Oriented Architecture.

COURSE RESOURCES

Course Resources

- MacCormack, A., Rusnak, J., & Baldwin, C. (2007). *Exploring the duality between product and organizational architectures: A test of the "mirroring" hypothesis*. [Working paper 08-039]. Boston, MA: Harvard Business School. Retrieved from http://www.hbs.edu/faculty/Publication%20Files/08-039_1861e507-1dc1-4602-85b8-90d71559d85b.pdf

- Van der Linden, F. J., Schmid, K., & Rommes, E. (2007). *Software product lines in action: The best industrial practice in product line engineering*. Berlin, DE: Springer. [Books24x7 version]. Available from http://common.books24x7.com/toc.aspx?bookid=31005

Glossary

Word	Definition
Abstract data type	A data type that is defined by the programmer and not built into the language. It is a grouping of related information that is denoted with a type, which allows data to be organized in a meaningful way.
Abstraction	A design principle that suggests that a concept in the problem domain should be simplified down to its essentials within some context.
Activities	Actions that further the flow of execution in your system.
Adaptation technique	A technique to realize variations in a product line system. In this technique, a component has only one implement, but it supplies interfaces to change or add on to it.
Application engineering	The development of products in a product line.
Architecturally significant requirements (ASR)	Requirements that arise from business drivers, which are considered significant to the system architecture.
Architecture stakeholders	A group of participants in ATAM who are stakeholders that want the architecture to successfully address the business needs but is not actively involved in the evaluation process. This could include end users, developers, and support staff.
Architecture Trade-off Analysis Method (ATAM)	A method for evaluating an architectural system that can be used by outsiders of a system.
Artifacts	A physical result of the development process.
Asynchronous	A type of request-response relationship where a client sends a request, but control returns right away, so it can continue its processing on another need.
Attribute refinements	More specific qualities of a system.

Availability	The amount of time the system is operational over a set period of time.
Ball connector	Part of a component diagram that displays a provided interface.
Behavioural design patterns	Patterns that focus on how objects distribute work and how they collaborate together to achieve a common goal.
Client	A program that requests a service or resource from another program (the server).
Client/server relationship	A relationship in which a client requests information or actions, and the server responds.
Client-host	The machine that hosts client software.
Clients	A stakeholder of software architecture who supply funding and outline the needs of the system.
Components	Independent, encapsulated units within a system.
Conceptual integrity	The concept of creating consistent software. Conceptual integrity requires making decisions about how a system will be designed and implemented, so it seems as if only a single mind guided all the work.
Concrete scenario	A concrete scenario is a quality attribute scenario that is used to characterize a specific system.
Conway's Law	A law that states that a system will tend to take a form that is congruous to the organization that produced it.
Creational design patterns	Patterns that tackle how to create or clone new objects.
Data accessor	Any component that connects to a database that can share data while operating independently, communicate with the database through queries and transactions, and save the new state of the system back into the database using transactions.
Data-centric software architecture	A software architecture that allows data to be stored and shared between multiple components, often through the use of a database.

Data flow architecture	A system architecture that considers the system as simply a series of transformation on sets of data.
Data integrity	The assurance that data is accurate and consistent over its lifespan.
Data persistence	The assurance that data will live on after a process has been terminated.
Database	A central repository for data.
Database management system (DBMS)	An automated system that manages and optimizes queries and transactions.
Decomposition	A major design principle of object-oriented modelling and programming. It takes a whole thing and divides it up into different parts. It could also take separate parts with different functionalities and combine them together to form a whole.
Designers	A subgroup of the evaluation team. Includes those involved in the architectural design of a system.
Domain engineering	The development of commonalities and variations in product lines. Essentially, the creation of the "infrastructure" of a product line.
Encapsulation	A fundamental design principle in object-oriented modelling and programming. Encapsulation involves three ideas: It bundles attribute values and behaviours that manipulate those values into a self-contained object; It also exposes certain data and functions of the object to other objects; or alternately restricts access to certain data and function to only within that object.
End users	A stakeholder of software architecture who will ultimately use the software product.
Evaluation Team	A group of participants in ATAM that includes designers, peers, and outsiders.
Event-based architectural style	A system architecture whose fundamental elements are events.
Event bus	The connector between event generators and consumers in the system.

Event generators	Also known as event consumers. Event generators send events, and event consumers receive and process these events.
Events	Includes signals, user inputs, messages, or data from other functions or programs. Events act both as indicators of change in the system and as triggers to functions.
Extension technique	A technique to realize variations in a product line system. In this technique, a common interface is provided for all the variations of the system. These are often called extensions, add-ons, or plugins.
Feedback loop	A situation where part of the output of a system is used for new input.
Feedforward	A situation where information from an upstream process can be used to control a downstream process.
Filters	Entities in pipe and filter architecture that perform transformations on data input they receive.
Flexibility	Indicates how interchangeable other modules are for a specific module. The more replaceable, the better the design.
General scenario	A general scenario is a quality attribute scenario that is used to characterize any system.
Generalization	A design principle that helps reduce the amount of redundancy when solving problems, by taking repeated, common, or shared characteristics between two or more classes and factoring them out into another class, so that code can be reused and characteristics can be inherited by subclasses.
Good alignment	When priorities of stakeholders in a software system match closely with priorities identified in a utility tree developed using the ATAM method.
Implicit invocation	An invocation where functions are not in direct communication with each other but where communication is mediated by an event bus.
Instance level diagrams	A deployment diagram that maps a specific artifact to a specific deployment target.

Interoperability	The ability of your system to understand communications and share data with external systems.
Interpreter	A component built into a system that can be used to run scripts and macros and to drive programmable actions specified by the user.
Kruchten's 4+1 View Model	The combination of logical, process, development, and physical views of a system along with scenarios that provide an understanding of a system's architecture.
Latency	The time it takes to produce an output after receiving an input.
Logical view	A view of a software system that focuses on functionality of a system and the objects within it.
Maintainability	Maintainability refers to the ease of modifying a software system or component in order to correct or improve faults, performance, or other attributes, or the ability to adapt to a changed environment.
Manifestation	A relationship where an artifact is a physical realization of a software component.
Modifiability	Modifiability refers to the ability of a system to cope with changes to functions, incorporating new functionality, or remove existing ones.
Nodes	Deployment targets that contain artifacts available for execution.
n-Tier architecture	Also know as multitier architecture. It is a layered architecture based on tiers.
Outsiders	A subgroup of the evaluation team. Includes those external to a project or organization.
Overhead	A combination of computation time, memory, bandwidth, processing, or other resources required to perform a task.
Package	A grouping of related elements in a software system, such as data, classes, or user tasks.
Package diagram	A UML diagram that shows packages and the dependencies between them in a system.

Packageable elements	Elements that can be packaged together, including data, classes, or even other packages.
Peers	A subgroup of the evaluation team. Includes those who are in the project but not involved in design decisions.
Performance	The system's throughput and latency in response to user commands and events.
Physical view	A view of a software system that focuses on the physical components of software, and how they interact and are deployed.
Pipe and filter architecture	A type of data flow architecture that has filters that perform transformations on data input they receive and pipes that serve as connectors for the stream of data being transformed.
Pipes	Entities in pipe and filter architecture that serve as connectors for the stream of data being transformed.
Principle of least knowledge	An underlying design principle for design patterns. It states that classes should know about and interact with as few other classes as possible.
Process view	A view of a software system that focuses on the implementation and hierarchical structure of the software.
Product line	Also known as product family. A grouping of products that usually share the same operating system, which allow for code re-use.
Productivity	Effectiveness of the effort of a team.
Project decision makers	A group of participants in ATAM who are project representatives with the authority to make project decisions. This group could include project managers, clients, product owners, software architects, and technical leads.
Project managers	A stakeholder of software architecture who plan, identify, and mitigate risks, and manage the project.
Quality	The degree of excellence of the software.

Quality attribute scenarios	A scenario used to determine if a system is able to meet the requirements that were set for the quality attributes.
Quality attributes	Measurable properties of a system used to gauge a system's design, run-time performance, and usability.
Reference Architecture	An architecture for a product line that products can build on or change so that each product does not need to build an architecture from scratch.
Replacement technique	A technique to realize variations in a product line system. In this technique, there could be a default component that is replaced with alternatives to realize variation. There may not even be a default; instead there is a gap in the system, and the developers must supply one of the components.
Request-Response	The communication between a server and client, where a client requests information or actions, and the server responds.
Reusability	Reusability refers to the ability to use existing products and by-products of software development, including code, designs, and documentation, more than once in the software development process.
Sandboxing	When different layers of a layered system run at different levels of authorization or privilege.
Scenario	Use cases or tasks required by end users of a software system.
Script	A description of the sequences of interactions between objects and processes.
Security	The system's ability to protect sensitive data from unauthorized and unauthenticated use.
Semaphore	A variable or abstract data type that toggles between two values: available and unavailable, which is used to control access to shared data.
Sensitivity points	Identifies processes in a system that could affect the specific quality attributes of a system relative to an ASR.

Separation of concerns	A principle of software design, which suggests that software should be organized so that different concerns in the software are separated into different sections and addressed effectively.
Server	A program that provides a service or resource upon request from another program (the client).
Server-host	The machine that hosts server software.
Socket connector	Part of a component diagram that displays a required interface.
Software Architecture	An aspect of the software development process, which examines the high-level aspects of a system.
Software developers	A stakeholder of software architecture who develops and creates the software.
Specification level diagrams	A deployment diagram that provides an overview of artifacts and deployment targets, without referencing specific details like machine names.
Stakeholders	The people who have an interest in software, either through use of the system or benefit from it.
Structural design patterns	Patterns that describe how objects are connected to each other.
Structured Query Language (SQL)	A language used by relational databases that allow queries to be asked of databases for information as well as allow transactions to be performed.
Synchronous	A type of request-response relationship, where a client sends a request, then awaits the server's response before continuing execution.
Testability	Testability refers to how easy it is to demonstrate errors through executable tests.
Throughput	The amount of output produced over a period of time.
Tier	Components that are on different physical machines. Often used interchangeably with layer, but this is not correct.
Trade-offs	An occurrence when one quality must be sacrificed or restricted for improvement in another.

UML activity diagram	A UML diagram that represents the control flow from one activity to another in a software system.
UML component diagram	A UML diagram concerned with the components of a system.
UML deployment diagrams	A UML diagram used to visualize deployment details of a software system.
Usability	The ease at which the system's functions can be learned and used by the end users.