Software Architecture

Status Completed

Lecture 7: UML Architecture Diagrams

03/06/24

Architecture overview and process

- Software architecture is how a software system is constructed at the highest level (the “system” level).
  - It defines the elements of a system, and what function(s) each element has, and how those elements relate to each other.
- Software architecture increases productivity by coordinating work and grounding discussions about a software system.
- Well designed, quality software architecture makes a system more maintainable, reusable, and adaptable.
- Stakeholders are people or groups who have an interest in a software system.
  - This includes software engineers, project managers, clients, and end users.
  - A clear software architecture communicates confidence to all stakeholders, most importantly clients who provide funding.
  - It is important for software architects to keep many perspectives in mind, and operate at many levels of abstraction.
    - This means covering code quality requirements at a deep level.
    - This means meeting the needs of both software engineering teams and the business the software system is being built for.
    - This means creating value (functionality, maintainability, longevity) for clients of a software system.
    - This means balancing monetary costs and time to deploy.
- Software architects should be wary of adopting new and emerging technologies, as there is no way to guarantee that those technologies will be suitable for future needs.
  - If the decision is made to implement a newer technology, requirements should be made to replace it with a more long-standing technology if need be.
- Agile development is implemented in software architecture by adjusting architecture as needed, in smaller cycles than in a waterfall architecture.
- Software architecture is often communicated through UML diagrams.

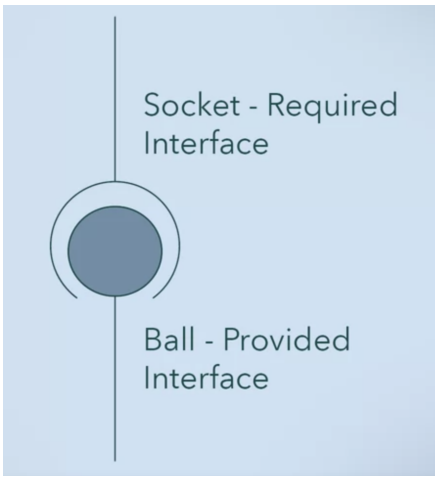
Kruchten’s 4 + 1 model view

- It is difficult to capture the complete functionality of a software system with a single perspective.
- There are many different ways to assess a software system:
  - The logical view assesses a software system through functionality, or how closely it meets a client’s requirements.
    - It focuses mostly of achieving the functional requirements of a system.
    - UML class and state diagrams are created to establish the vocabulary of the problem, and the resulting system.

        - It defines all the classes, as well as their attributes and behaviors.
    - The process view assesses a software system through its performance and scalability.
        - This view would show the execution order of different objects in a system.
        - It also describes behaviors that are concurrent or asynchronous.
        - UML sequence and activity diagrams are useful for expressing the process view.
    - The development view assesses a software system through its specific implementation.
        - This view is highly influenced by the hierarchical structure of a system, and the programming language(s), libraries, and tools used to build the system.
        - This view also includes scheduling, budgeting, and assignment delegations.
    - The physical view assesses how a software system interacts with physical components, such as databases.
        - It describes how the other views "map" to hardware that is used to run the software system.
            - This can be done through a UML deployment diagram.
    - All of these views are influenced by the use-cases/scenarios for end users of a software system.
        - These scenarios describe how the four other views work together.
        - A script is used to show how the objects and processes in a system interact.
- These views and use-cases/scenarios form Kruchten's 4 + 1 view model, a method of understanding any system architecture deeply.
    - If two or more of these views are similar enough, they can be either be consolidated, or one or more can be omitted.

UML component diagram

- UML component diagrams are concerned with the components of a system.
    - Components are independent, encapsulated units within a system.
    - Relationships describe interactions between components
    - Component diagrams concern themselves with high level interactions in a system, not the low level methods, attributes, and implementations of individual classes.
- Ball connectors denote a provided interface.
    - A provided interface shows that a component *provides* an *interface* for other components to interact with it.
- Socket connectors denote a required interface.
    - A required interface shows that a component expects a certain interface.
- An assembly relationship is where one component's provided interface matches another component's required interface.

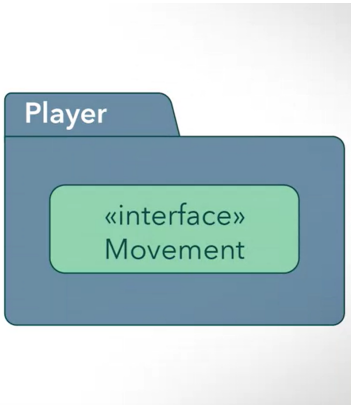


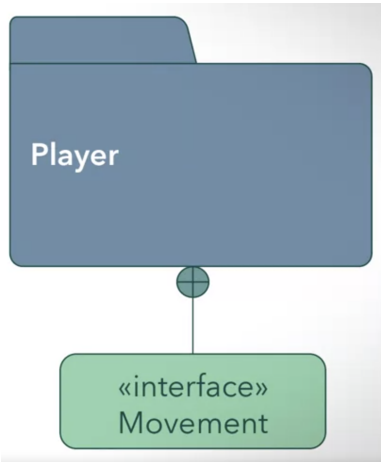
- There are multiple steps that should be taken when building a component diagram:
  1. Identify the main objects used in the system.
  2. Identify the relevant libraries needed for the system.
     - Any third-party libraries or systems should be denoted as such on component diagrams.
  3. Identify relationships between the individual components.

03/07/24

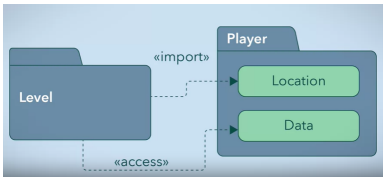
UML package diagram

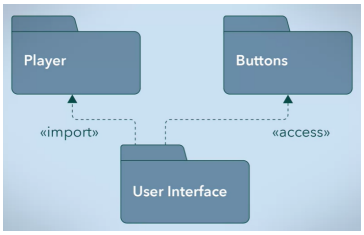
- UML package diagrams express packages and the dependencies between them.
  - Packages group together related elements in a software.
  - Packages create a namespace, where elements of that package are organized into a separate scope under the package name.
  - An element can be identified through a qualified name, that being its own name, as well as the name of the package it is in.
- Each package is denoted by a folder icon.
  - Elements in a package are either nested or connected to the package.
  - Elements that are accessible outside the package are denoted with a `+`, and those that are not with a `-`.

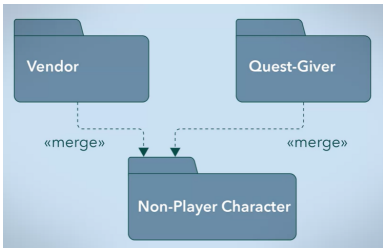




- Packages can import elements from other packages, whole packages, or merge with other packages.



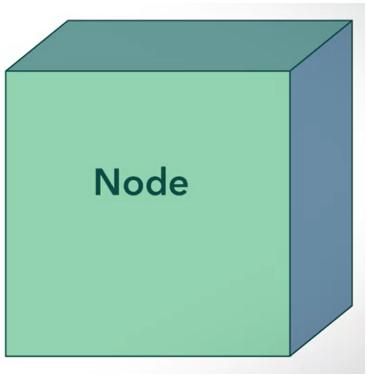


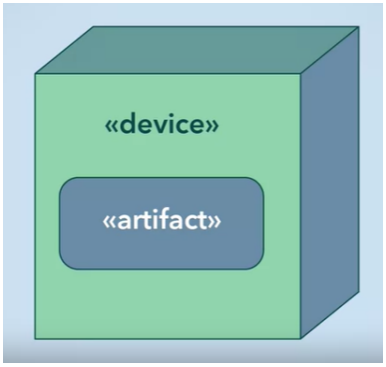


- Package diagrams are useful for understanding the relationships between packages and namespaces.
- Showing a system visually makes spotting flaws and inconsistencies much easier.

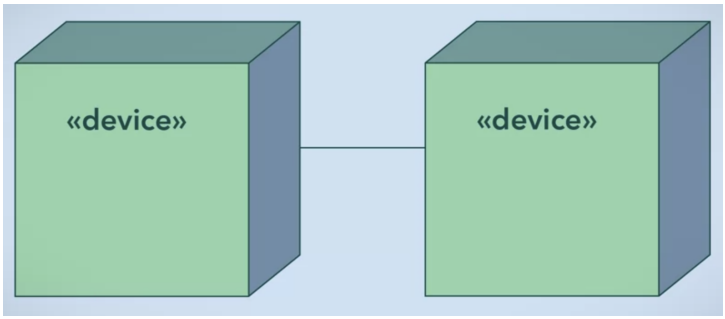
UML deployment diagram

- UML deployment diagrams are used to gain a comprehensive understanding of what files and executables are needed to deploy a software system.
  - The two types of deployment diagrams and specifical level diagrams and instance level diagrams.
    - Specification level diagrams give an overview of artifacts and deployment targets.
      - In other words, it gives a general overview of deployment.
    - Instance level diagrams map specific artifacts to specific deployment targets such as machines or hardware devices.
      - In other words, it expresses a more specific look at deployment.
- Artifacts are physical results of development processes like executables, libraries, and assets.
- Nodes are deployments targets, which contain artifacts available for execution.
  - Nodes and hardware devices are denotes with cubes.

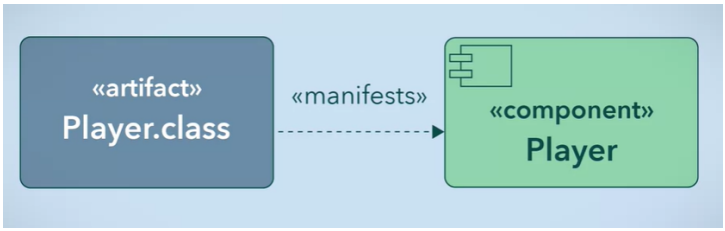




- A communication path between nodes is denoted with a line.

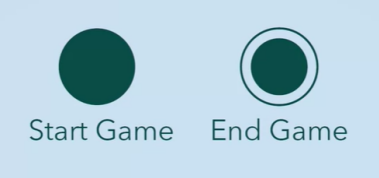


- A manifest represents a physical realization of a software component.



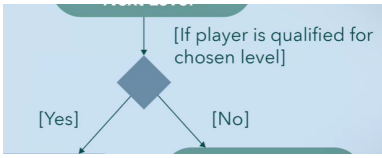
UML activity diagram

- UML activity diagrams shows the control flow between different activities in a software system.
  - Activities are actions the further the flow of execution in a system.
  - In a sense, an activity diagrams captures the dynamic behavior of a software system.
- To create an activity diagram one must first identify the activities, or the actions performed by the system.



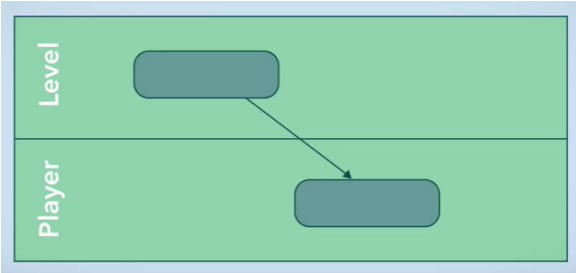
- Start and end nodes are denoted by labelled circles.

- Intermediate activities are denoted with pill shapes.



- Decision nodes are denoted by diamonds.

- Activity diagrams can also express concurrent activities that occur in parallel.
  - These are denoted with partitions, which divide activities up into different categories based on their role in the software system.



▼ Lecture 8: Architectural Styles

03/07/24

Abstract data types and the object-oriented paradigm

- The programming paradigm chosen to build a software system will effect the architectural style of the software system.
- The object-oriented architectural style results for the object-oriented programming paradigm.
  - This architectural style makes use of object-oriented design principals and design patterns.
  - While this paradigm makes the most sense for some software systems, it is not always the best solution.
- One should consider what data is handled by a software system, and how that data can be broken down into abstract data types.

03/08/24

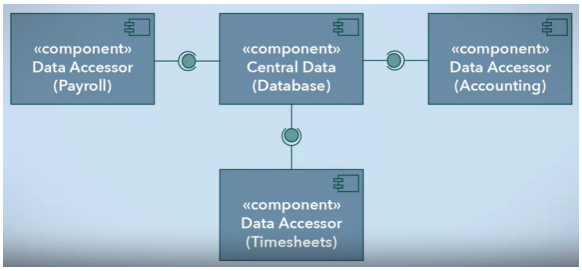
Main program and subroutine paradigm

- The main program and subroutine paradigm is a style that is fundamentally focused on functions.
  - A program is broken into a main function with subroutines.
  - Routine, procedure, and function are all synonyms for subroutine.
- The structure of the subroutines are hierarchical, and therefore the program can be modelled as a directed graph of function calls.

- There is always one entry point and one exit per subroutine, which makes the control flow of the subroutine easier to follow.
- The procedural programming paradigm promotes function modularity and reuse by building programs from smaller subroutines.

Databases

- Data-centric software architectures allow data to be stored and shared between multiple components.
  - This is usually done by incorporating a database into the overall system design.
    - Databases ensure data integrity, that data is accurate and consistent over its lifespan.
    - Data bases ensure data persistence, that data is not lost even after a process has been terminated.
- Central data is a component used to store and serve all data across components that may need it.
  - This component serves data as requested by the data accessors.
  - This component is passive, as it is usually not involved in logic or data processing.
    - Central data is primarily concerned with storing and serving information.
- Data accessors are components that connect to the central data component.
  - They should be able to share data while operating independently.
  - They communicate with the database through queries and transactions.
    - Queries "ask" a database to get information.
    - Transactions "tell" a database to perform an operation.
  - Different concerns can be made into separate data accessors.
    - Separate data accessors allow a software system to be scaled easier.

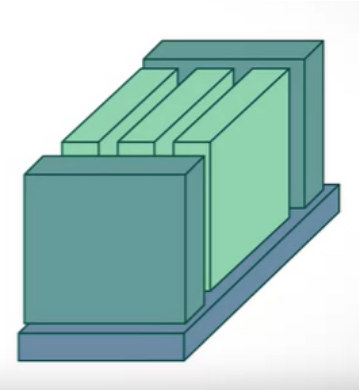


- Data-centric architectures also have disadvantages.
  - The system becomes heavily reliant on the central data component or repository.
    - Data redundancies are used to alleviate some of this risk.
  - The physical infrastructure needed can be extremely expensive.
  - It is difficult to change the existing data schema without completely overhauling the system.

Layered systems

- Layers are collections of components that work together toward a common purpose.

    - Layers only interact with themselves and their adjacent layers.
        - This shallow interactions follows the principal of least knowledge.
- This layering allows for a separation of concerns between layers.
- Layered architectures create more loosely coupled software systems.
    - When replacing a single layer one only needs to make sure it interacts with its adjacent layers correctly.
- One disadvantage of layered systems is that layers cannot interact with any non-adjacent layers, which may prevent certain desired functionalities.
- Layered systems may use three dimensional representations, which allow for more complex adjacencies.

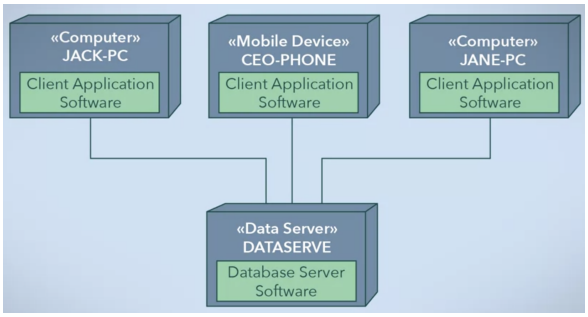


Client-server n-tier architectures

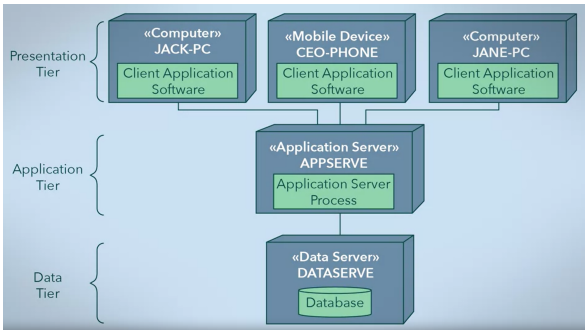
- Tiers a similar to layers, but often refer to components that are on different physical machines.
    - These systems are called n-tier or multitier systems.
- The relationship between two tiers in this architecture is often a client-server relationship.
    - The server provides services such as databases or data processing.
    - The client requests the server's services through messages.
    - The communication between the client and server is called request-response.
- The request-response relationship can be synchronous or asynchronous.
    - For a synchronous request, the client awaits the server's response before continuing execution.
        - This is denoted with a closed arrowhead on sequence diagrams.
    - For an asynchronous request, control to the client is returned immediately so that it can continue execution while awaiting the server's response.
        - This is denoted with a line arrowhead on sequence diagrams.

```
sequenceDiagram
    MediaPlayer->>MediaServer : view media library (request, synchronous)
    activate MediaPlayer
    activate MediaServer
    MediaServer--)MediaPlayer : media library index (response, synchronous)
```

```
    deactivate MediaServer
    deactivate MediaPlayer
    MediaPlayer-)MediaServer : play selection (request, asynchronous)
    activate MediaPlayer
    activate MediaServer
    MediaServer--)MediaPlayer : video stream (response, asynchronous)
    deactivate MediaServer
    deactivate MediaPlayer
```



- A simple client-server relationship with one client and one server is called a 2-tier architecture.



- A 3-tier architecture inserts an application tier between the server and client tiers.
- This middle layer determines how and when data can be changed in what ways.

- Additional tiers can be added based on each unique software system's needs.
- These multi-tier architectures are highly scalable, and gain benefits through centralization.
  - Data or computing power only needs to reside on one server, but can be accessed by any other machine.
- The tight coupling due to the relationships between different tiers makes the system difficult to change and maintain.
  - Like databases, the server also acts as a central point of failure.
  - Redundant servers are possible, but add additional complexity.

Interpreters

- Interpreter-based architectures allow end users to interact with a software system through software scripts.

> 🧭 For more information on interpreters, the notes from Programming Languages, Part B can be referenced.

- Interpreters can be used to port a software system to different platforms, making the system platform independent.
- Interpreter can provide more flexibility, but may be slower than other alternatives due to their line-by-line evaluation behavior.
- Java programs are compiled into an intermediate language, byte-code, that is loaded into the Java Virtual Machine (JVM), which executes this byte-code.

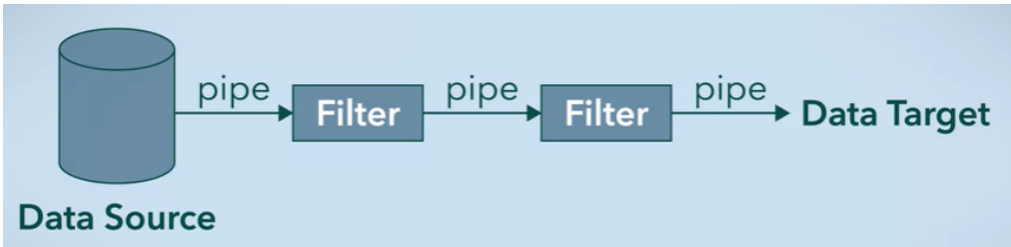
    - Frequently used expressions are translated into machine code and executed immediately.
    - Infrequently used expressions are executed by the JVM interpreter.
- The JVM allows Java to be run on may different environments.

State transition systems

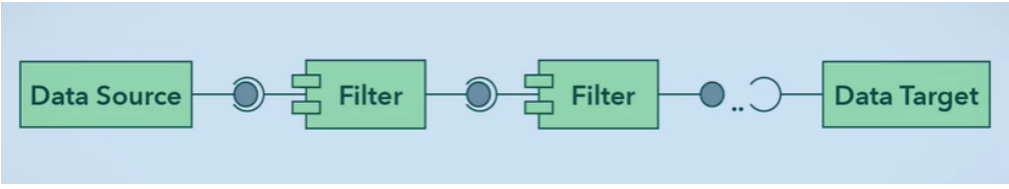
- State transition systems describe all the potential behavior of a software system.
    - They describe how each state can be arrived at, and the behavior of the system in that state.
- State transition systems are important, as they help software architects understand how different states can be achieved.
- State describes the information in a system at a single moment.
- Transitions describe the change in a system from one state to another.
    - Many systems have multiple transitions to and from each state, making the behaviors of a system unpredictable or non-deterministic.
- Behavior describes what a system is expected to do in a specific state, or in response to an event or condition.
- State transition systems can be expressed as directed graphs.
    - Each node of the directed graph represents a state.
    - Each edge of the directed graph represents a transition.
    - One can determine how a system reaches a specific state by traversing the graph.

Pipes and filter architecture

- The pipe and filter architecture is a type of *directed* data flow architecture.
    - Data flow architectures consider systems as a series of transformations on sets of data.
- Filters perform transformations on data inputs they receive.
- Pipes serve as connectors for the streams of data between filters.



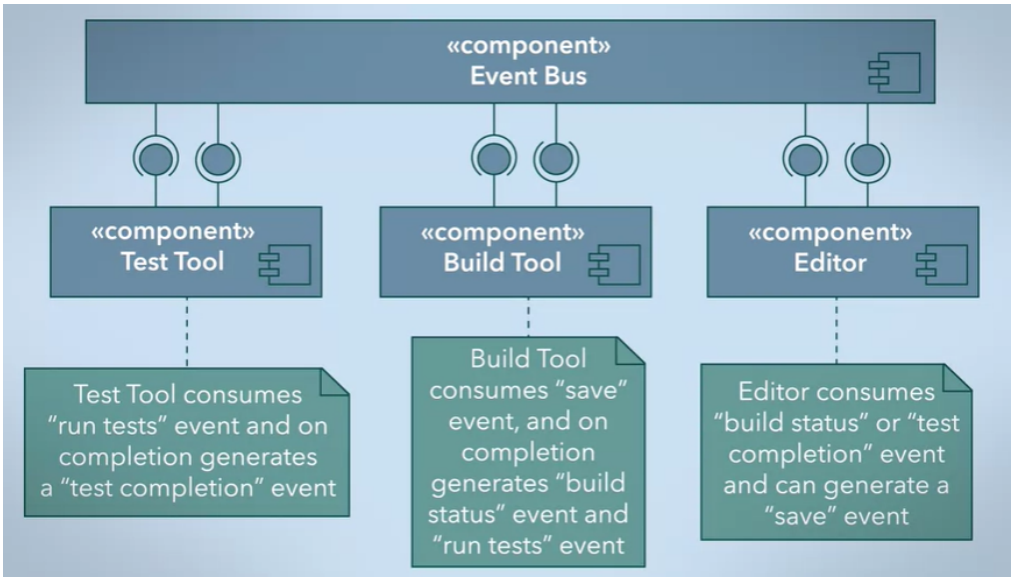
- The order in which the filters transform data has an effect on the resulting data target.
- UML component diagrams are often used to represent pipe and filter architectures.



- Pipe are filter architectures ensure loose and flexible coupling between filter components, as each filter operates independently of other filters.
- Pipe and filter architectures have some disadvantages.
    - Certain filters may create bottlenecks in the data processing pipeline.
    - This architecture is very linear, and therefore cannot be used for interactive applications.
        - Data processing can also take time, and therefore is not suitable for systems that require quick responses.

Event based architecture

- Event based architectures respond to events.
    - Events can be signals, user inputs, messages, or external data.
        - They indicate change and trigger functions in a system.
- There are two specific types of events.
    - Event generators send events to event consumers.
    - Event consumers receive events and process them.
    - A function can be both an event generator and consumer.
- Event-based functions experience implicit invocation, that is, event-based functions are not in direct communication with each other like they would be in, say, a subroutine architecture.
    - All communication between functions are mediated by an event bus.
    - Event generators and consumers are bound to the event bus, and must pass through it.
    - When the event bus detects a specific event, it distributes that event to all appropriate event consumers.
    - The Observer design pattern is a manifestation of this event-based architectural style.
- One way to implement this architecture is to have a main loop that continuously “listens” for events.
    - This main loop is an event generator that can implicitly invoke event consumers.



- Because event generators and consumers do not need to be aware of each other's existence, this creates an architecture will very loose coupling.
- Two or more event consumers can be running concurrently on the same data with this architecture.
  - This can increase the efficiency of the software system.
  - This can lead to race conditions, where the shared data may not be updated in the correct order.
    - This may be solved with a binary semaphore, which toggles between *available* and *unavailable*.
    - These semaphores indicate whether shared data is in use or not, and can prevent concurrent data processing.
      - Functions can check if data is available by accessing these semaphores, and wait if the data is unavailable at the moment.
      - If the shared data is available, the function accessing it toggles the semaphore to unavailable before processing the data.
- Event based systems do not execute in a predictable way.
  - Due to this asynchronous behavior, the control flow depend entirely on what events occur during runtime.
  - This makes event based systems ideal for interactive applications that rely on user input.
  - Systems that use this architecture are able to evolve and scale easily.

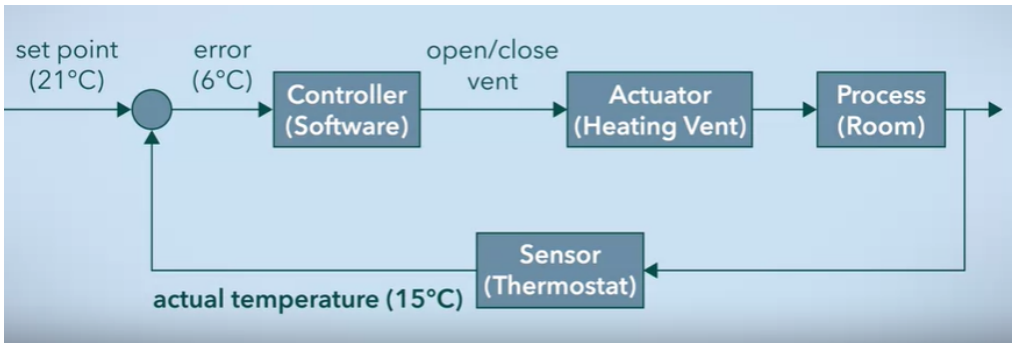
Publish-subscribe architecture

- Publish-subscribe architectures are similar to event based architectures.
  - A component can be either a publisher or subscriber, which rely on implicit invocation.
  - The difference from event based systems is that subscribers register their interest to publishers through callbacks.
- This architectural style is ideal for systems where each publisher has a relatively small number of subscribers.
  - Because publishers make a call to each individual subscriber, publishers can take on a lot of processing work as the number of subscribers grow.

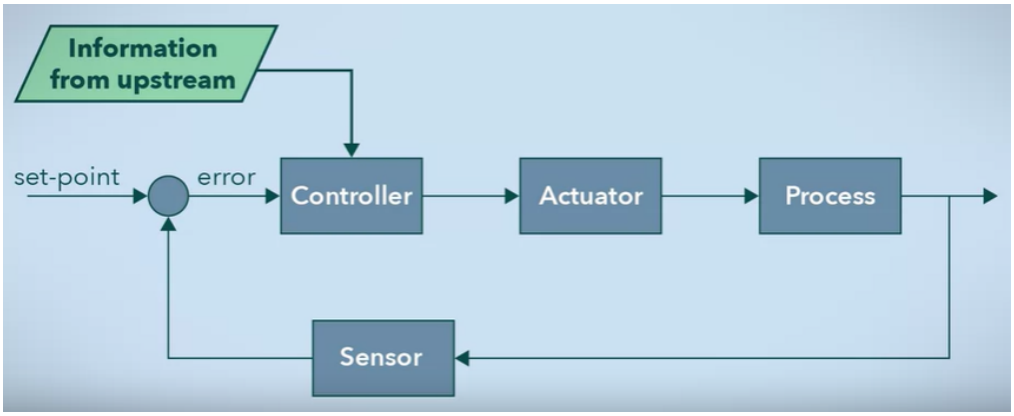
- Connectors are introduced as intermediaries to distribute and filter messages to subscribers.
  - These connectors could be event busses, network protocols, etc.
    - If the connector is an event bus, the resulting architecture is essentially an event based architecture.
      - The main difference is that publishers can still only *send* messages, and subscribers can still only *receive* messages.
      - A component cannot be a publisher and a subscriber at the same time.
  - Connectors relieve work for the publisher.

Process control

- The most basic form of process control is called a feedback loop.
  - Feedback loops adjust a process based on some information.
- Feedback loops have four main components:
  1. Sensors monitor information.
  2. Controllers determine the logic of the feedback loop.
     - The controller logic is run continuously, as the process is always changing.
  3. Actuators manipulate the process.
  4. Processes are what is being controlled.



- The software compares a measured value to a setpoint constant, and controls the process to achieve the desired result.
- Feedback loops are closed loops, since information from the process is used to control itself.
  - Open loops are processes that are controlled without monitoring that process.
    - Open loops are simpler, but cannot deal with changes in the process.
    - These loops usually lack sensors that can check if a process is succeeding.
- Feed forward control is when information from an upstream process is used to control a separate, downstream process.



▼ Lecture 9: Architecture in Practice

03/08/24

Quality attributes

- It is important to consider practical architectural questions such as:
  - What makes an architecture good or bad for a given problem space?
  - How can you identify and architecture's quality, and how one can improve its quality?
  - How can architecture be used to plan development and structure a development team?
- Design patterns cannot be used to assess the quality of an architecture as they only address functional concerns, not business needs and concerns.
- Therefore software architectures address the larger scope of the system, including functional *and* non-functional concerns.
  - The quality of a software architecture is based on how well it addresses a set of requirements for a specific use case in specific environment, and therefore cannot be labelled "good" or "bad" out of that context.

Concerns design patterns address

- Maintainability
- Reusability
- Performance

Concerns software architectures address

- Testability
- Usability
- Availability

- Non-functional requirements are not always clear or explicit, and may vary across stakeholders.
- The quality of a system is measured by quality attributes.
  - Quality attributes are measurable properties of a system used to gauge a system's quantifiable attributes.

Quality Attributes (Developer Perspective)	
Quality Attributes	**Measurement Of**
Maintainability	The ease at which your system is capable of undergoing changes.
Reusability	The extent in which functions or parts of your system is used in another.
Flexibility	How well a system can adapt to requirements changes.
Modifiability	The ability of a system to cope with changes, incorporate new, or remove existing functionality.
Testability	How easy it is to demonstrate errors through executable tests.
Conceptual Integrity	The consistency across the entire system, such as following naming conventions.

Quality Attributes (User Perspective)	
Quality Attributes	**Measurement Of**
Availability	The amount of time the system is operational over a set period of time.
Interoperability	The ability of your system to understand communications and share data with external systems.
Security	The system's ability to protect sensitive data from unauthorized and unauthenticated use.
Performance	The system's throughput and latency in response to user commands and events.
Usability	The ease at which the system's functions can be learned and used by the end users.

- 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.
- One mark of a high quality system is having detailed and up to date documentation of the system.
    - This ensures that the details of a system are not lost in the collective knowledge of a team over time.
    - Therefore, knowledge of a system should not reside in a single person.
- Having a technical lead for the system design process will ensure all technical details are caught before they waste a development team's time and resources.

Analyzing and evaluating architecture

- All quality attributes use quality attribute scenarios to determine if a system is able to meet the requirements for that quality attribute.
- There are two types of quality attribute scenarios:
  1. General scenarios are used to characterize any system.
     - In a general scenario, only high level events are considered.
  2. Concrete scenarios are used to characterize a specific system.
     - Events specific to the system are considered.
       - This allows specific stimuli to be tested in specific environments, to measure how well the system responds.
- It is system's inability to handle unexpected failures, that stop it from achieving a quality attribute.
- The Architecture Trade-off Analysis Method (ATAM) enables evaluators of a system to not be familiar with the architecture of it or the problem space.
  - There are three groups of participants in ATAM:
    1. The evaluation team consists of designers, peers, and outsiders.
       a. Designers are those involved with the architectural design.
       b. Peers are those who are part of the project, but are not involved with the design decisions.
       c. Outsiders are those who are external to a project or organization.
          - This viewpoint helps eliminate any bias towards the project.
    2. Project decision makers are project representatives with the authority to make project-based decisions.
    3. Architecture stakeholders are those who want to architecture to address business needs, but are not directly involved in the architecture evaluation process.
  - This a common approach for analyzing and evaluating system architectures.
    - It does not require important knowledge of the system.
    - It puts emphasis on the stakeholders for a system.
    - It focusses on a system's quality attributes.
- Knowing the risks, sensitivity points, and trade-offs of a system is important to producing a high quality system.

Relationship to organizational structure

- Conway's law states that a system will tend to take a form that is congruous of the organization that produced it.
  - Studies suggest that looser organizational structures like open-source projects, lead to more loosely-coupled and modular code.
  - This also means that tighter organizational structures like professional development teams tend to create more tightly-coupled code.
- Conway's law can be used to plan development teams around the desired architecture.
- Although Conway's law is not an exact science, it helps to be aware of it and its implications for the resulting architecture of a system.

Product lines and product families

- When developing programs, one should always be looking for opportunities to re=use code.
  - Even if reusable code need to be modified or adapted, it is still faster than writing code from scratch.
- One method for efficient development is to treat a group of products like a product line or family.
  - Product line and product family are interchangeable terms in most contexts.
- One reason to develop a product line is that is saves costs and reduces its time-to-market.
  - Reusing code across products means less development per product.
- Products lines enhance user experience by reducing or removing the learning curve for end users of a system.
- The book *Software Product Lines in Action* discusses the architectural considerations necessary for software product lines.
  - One should separate the features that are constant across products from the features that vary.
    - Commonalities are the features of a product line that stay the same in every product.
    - Variations are the features of a product line that vary between products.
    - Product-specifics are the features that are specific to one and only one product.
- Product line development teams are generally divided into two categories:
  1. Domain engineering describes the development of commonalities and variations.
     - This team often designs a product with future products in mind.
  2. Application engineering describes the actual development of a product.
  - Separation of these teams allows for separate development cycles that do not wait on each other as heavily.
- New products in a product line commonly build upon a base reference architecture, which cuts down on initial development time.
  - Therefore, the reference architecture must include the capacity for variation.
    - Reference architectures may consist of one implementation, but supply interfaces to modify or add additional functionality.
    - Reference architectures may be able to be extended through a common interface for all products.
      - These are most commonly manifested as extensions, add-ons, or plugins.