OOAD & Software Engineering (UE18CS353)
Unit 2

Aronya Baksy

March 2021

1 Software Architecture and Design

- **Software Architecture**: high level decomposition of software into components and characterization of the interaction between the components
- **Software Design** involves further decomposition of components, as well as identifying detailed functionality and attributes of each component.
- Design involves implementation details:
  1. Data structures/algorithms used for actual implementation
  2. User interface details for interaction with system
  3. Design patterns to be used (structural/behavioural/creational)

1.1 Attributes of Software Architecture

- Address all stakeholder perspectives, separate stakeholder concerns
- Realize all use cases and user scenarios defined at the requirement phase
- Drives quality attributes (non-functional reqs like availability, security)
- Overall vision of system functionality that is separate from actual implementation detail

1.2 Factors that influence Software Architecture

- Functionality of the system
- Legal and regulatory obligations
- Constraints on cost, skill levels of staff, technology available
- Audience, user environment/experience that is planned, usage characteristics

1.3 Software Design

1.3.1 Enablers

- **Abstraction**: focus on essential properties, hide implementation details as far as possible
- **Modularity**: Extent to which a large module can be decomposed.
  - **Coupling** is the extent to which modules are interdependent
  - **Cohesion** is a measure of the degree to which the elements of the module are functionally related (a measure of how focussed are the elements of a single module towards accomplishing a single task)
  - Ideal properties are low coupling and high cohesion

- **Information Hiding**: Make use of the need-to-know principle, as well as encapsulation, separation of interface and implementation
- **Simple Design**: both inter-module and intra-module, limit complexity of modules
- **Hierarchical structure**

1.3.2 Issues to be handled

- Concurrency, event handling
- Distribution and organization of components
- Non-functional requirements (fault tolerance etc.)
- Error handling, exception handling
- Interaction, presentation
- Data presentation

1.4 Differences between Software Architecture and Design

Architecture	Design
Big picture decisions on frameworks, tools, languages, scope, goals and high level methodologies	Small picture decisions on implementation, programming idioms, code organization, design patterns, local constraints on requirements
Abstract decisions on strategy, structure and purpose	Tactical decisions on implementation, practice (more concrete)
Structure of the system, software components, visible properties of those components, and the relationships between them	Problem-solving & planning for a software solution internal to the system
High impact and tough to change	Lesser (more localized to one stage) impact and easier to change
More influence on non-functional requirements	More influence on functional requirements

1.5 Approaches to decomposition

1.5.1 Layering Approach

- Organize software into layers that are dependent on each other in a sequence, each layer solving one task
- Allows for easy customization, fault isolation, acceptance testing and easy development

1.5.2 Distribution between Computing Resources

- Each process in a system has its own thread of control (and associated resources)
- Used when multiple clients need access
- Allows greater fault isolation, more separation of concerns, redundancy which allows high availability

1.5.3 Exposure

- Based on what services are exposed and what services are consumed by a component

1.5.4 Functionality

Based on problem domain and the logical separation of functionalities (e.g.: modules for login, customer service, inventory)

1.5.5 Generality

- Generalize components to be reused in other applications or other modules

1.5.6 How to Decompose?

- Divide and conquer
- Stepwise refinement (start simple, iteratively refine to get final module)
- Top-down approach: Start from a top-level overview, then detail each component
- Bottom-up approach: Specify individual components and compose them into modules
- Information hiding using encapsulation, separation of implementation and interface

2 Design Methods

2.1 Data Flow Method

- A two-step process, consisting of:
  1. **Structured Analysis** (SA) outputs aogical design, drawn as a set of data flow diagrams
  2. **Structured Design** (SD) transforms the logical design into a program structure represented as a set of structure charts
- A **data flow diagram** (DFD) has the following components:
  - **External Entity**: Source/Destination of a transaction, outside the domain of the DFD. They are represented as rectangles
  - **Process**: Transforms data and gives transformed output. Represented as a rectangle
  - **Data Store**: Represents data storage and its manipulation. They lie between processes, represented with parallel lines
  - **Data Flow**: Direction of data transfer between processes, entities and data stores. Represented using arrows.
- Decomopse DFD into more granular and low-level diagrams, then build a **minispec** (minimized description of processes in the DFD for communication of the algorithm)
- The contents of the minimal decomposed DFD are maintained in a **data dictionary**. Data dictionary symbols are:
  - []: Include one of the enclosed options
  - +: AND
  - |: Separate options
  - (): Enclosed options are optional
- Sample data dictionary:
  - borrow-request = client-id + book-id
  - return-request = client-id + book-id
  - log-data = client-id + [borrow | return] + book-id
  - book-id = author-name + title + (isbn) + [proc | series | other]
- The SA phase outputs the DFD, minispec and data dictionary. The SD phase takes these as input and transitions them into the hierarchical structure charts
- Heuristics used for this transition are based on coupling and cohesion. Most commonly used heuristic is *transform-centered*
- In this heuristic, the procedure is:

  - Trace input through the DFD until it is no longer considered as an input, same for output
  - The in-between bubble is considered as the transform
  - Processes in DFD become components of structure chart. Data flows in DFD become component invocations (the arrows are in the opposite direction in the two diagrams)

2.2 Booch's Method (for OO Development)

- Consists of an Object Modelling Language (currently most widely used is UML, Unified Modelling Language), an iterative development process and a set of recommended practices
- The **macro process** consists of:
  - **Conceptualization**: Identify core functional requirements,
  - **Analysis**: Develop a model of the desired behaviour
  - **Design**: Create architecture
  - **Evolution**: for implementation
  - **Maintenance**: post delivery
- The **Macro** process consists of:
  - Identify classes, objects
  - Identify semantics and relationships between the above
  - Specify all interfaces and their implementation

3 Architectural Views, Styles and Design Patterns

Architectural View	**Architectural Style**	**Design Pattern**
Ways of describing Software Architecture to all Stakeholders from different perspectives	Organization and structuring of subsystems and components	Known, proven solution to a design problem that organizes subsystems
UI View, Process View	Client-Server, Pipe-filter, P2P	MVC

3.1 Architectural Views

3.1.1 Module View

- Structure the system as a set of modules (each having some functional responsibility)
- An Architect enumerates what each unit will have to do and assigns each unit to a module
- Larger modules may be decomposed into smaller units for easy change management
- Used primarily for organization and documentation, less emphasis on actual runtime environment

3.1.2 Component and Connector View

- Runtime view of the software, dynamic
- Components (processing elements) are software elements that transform data from input to output (either computation, or server that responds to requests, or a controller that governs a sequence of events)
- Data elements store information that is to be processed
- Connecting elements connect the processing and data elements logically or physically (e.g.: RPC)

3.1.3 Allocation View

- **Deployment Structure** shows how the software components are assigned to the available hardware in terms of two relationships:
  - The "allocated-to" relation indicates which hardware element contains the particular software element
  - The "migrates-to" relation indicates dynamic allocation of hardware elements

  This structure allows one to reason about attributes such as performance, consistency, and availability. It is of particular use in distributed/parallel systems
- The **Implementation Structure** indicates how software is mapped onto file structures in the system's development, integration, or configuration control environments
- **Work Assignment Structure** shows who is doing what and where (indicates functional commonalities within a team)

3.1.4 Krutchen's 4+1 View

- **Use case view**: exposing the requirements of the system or the scenarios
- **Design view**: Exposes vocabulary of the problem space and the solution space using Class Diagrams, Sequence diagrams etc
- **Process view** encompasses the runtime behavior of the system (in terms of threads and processes). Addresses performance, Concurrency etc.
- **Implementation view**: Addresses the realization of the system. UML diagrams like package diagrams are used
- **Deployment view**: Focuses on system engineering issues

3.2 Architectural Styles

- A pattern for organizing the components in an architecture, characterized by the notable features (addresses structure and behvaiour of system)
- Architectural Styles provide:
  - **Vocabulary**: Set of design elements to be used (e.g.: client, server, database, etc.)
  - **Design Rules**: Dictate connection of processing elements
  - **Semantic Interpretation** of the connected design elements
  - **Analysis** such as deadlock detection/scheduling

3.3 Architectural Patterns

Architectural Patterns deal with naming and organization of layers in a layered software

3.3.1 Single Layer Architecture Pattern

- Also referred to as monolithic architecture
- Consists of a single application layer that supports the user interface, the business rules, and data processing
- e.g.: MS Word, client-server applications

3.3.2 2-Layer Architecture

- Approach 1: UI and business rules on one system, data storage and processing on another
- Approach 2: UI on one system, business rules and data operations on another system
- Procedures on the server can be either explicitly called or implicitly called using triggers
- e.g.: SQL server

3.3.3 3-layer Architecture

- Display layer contains the UI, Logic layer contains business logic and State layer contains the data storage and processing system
- Clients never directly access the state layer

3.3.4 Client-Server Architecture

- The client is a single-user workstation that provides UI interface and presentation logic, along with a connection to the server
- The server is a high-powered industrial machine with shared memory that serves client requests (implements the business logic and database connections needed for the same)

3.4 Design Patterns

- A general, reusable solution to a commonly occurring problem within a given context in software design
- A named collection of architectural decisions or design approaches which results in a successful solution

3.4.1 OO Design Patterns

- Creational Patterns deal with object creation (e.g.: Singletion, Builder, Factory)
- Structural Patterns deal with composition of objects (e.g.: Adaptor, Bridge, Facade, Proxy)
- Behavioural Patterns deal with interaction of objects (e.g.: Interpereter, Iterator)
- Distributed Patterns deal with interfaces for distributed systems (e.g.: Skeleton, class stubs)

3.4.2 Singleton Pattern

- An example of a creational design pattern. Used when only one instance of a class has to be created and that object is the only access point for that class
- The single instance is accessed using a `getInstance()` method in the class. The `getInstance()` method creates the instance if not existing, or returns a reference to the existing instance
- Used for centralized management of global resource
- e.g.: Logging class, config class, window manager, DB connection manager

3.4.3 Procedural Patterns

- **Structural Decomposition Pattern** breaks down large systems into communicating smaller subsystems
- **Work Organization Pattern** defines how the subsystems communicate (e.g.: Client/Server, P2P)
- **Access Control Pattern** defines access to components and services ususally via a proxy

- **Management Pattern** defines how to handle homogeneous collections as a whole (e.g.: command processor, view handler)
- **Communication Pattern** defines how to organize communication across modules (e.g.: forward-receive, pub-sub)

3.4.4 Anti-Patterns

- Situations that are undesirable and best avoided are defined as anti-patterns
- In Agile approaches, refactoring is applied when anti-patterns are encountered
- e.g.: **God class** that holds too much responsibility, **Lava flow** is dead code that gets carried on for a long time

4 Class Diagram

Represents classes, relationships between classes, and attributes + operations of classes.

4.1 Object

- A concept/abstraction/something with identity that has a specific meaning towards an application
- They have an identity that is unchanged through their lifetime. Can be concrete or abstract
- Objects play different roles wrt one another. These roles include multiplicity

4.2 Class

- Group of objects that share similar properties, behaviours, relationships and semantics
- Classes have responsibilities that an object of that class should be able to fulfill
- Classes abstract objects, hence generalize from specific cases to smaller number of generic cases
- In the class diagram, a class is represented with a vertical set of stacked boxes:
  - Top box includes class name
  - Second box has attributes (name, datatype, visibility)
  - Third box has operations (methods, specified along with arguments, type of arguments and return type)
  - Bottom box has responsbilities of that class
- A responsibility is a contract or obligation of a class to perform a particular service (represented as text phrases)
- **Class Responsibility Collaborator** or CRC Cards are used for brainstorming during the low level design. Each member of the team writes their own CRC card and the results are combined
- Collaborator is the set of all other classes that a given class has to interact with in order to fulfill a particular responsibility

4.3 Relationships

4.3.1 Association

- Describes a relationship between instances of one class and instances of the same or another class represented as a connection
- e.g.: Customer *owns* Account, Student *enrolls for* Elective
- Multiplicity denotes the number of instances or objects of one class that can be associated with one instance of another class

- UML notatitions for multiplicity:
  - 1: one and only one
  - 0..1 : 0 or 1
  - M..N: between M and N, both inclusive
  - 0..* : 0 or more instances
  - 1..* : 1 or more instances
  - *: Any positive integer
- An association can have an associated **navigability** (i.e. direction) that is denoted with an arrow (e.g.: Router → DNSServer, here the router sends messages to the DNSServer to access its services, but the DNSServer has no knowledge of the router)
- Association classes are classes that are associated with relationships between 2 already existing classes (e.g.: Registration is associated with the relationship between Product and Warranty)
- Association relationships have 2 forms:
  - **Aggregation** is a strong association wherein a complex object is made of smaller less complex objects (e.g.: Car is made of Engine, Gearbox, Chassis etc.)
  - Aggregation specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate.
  - Aggregations are denoted by a hollow-diamond adornment on the association.
  - **Composition** indicates a strong ownership and coincident lifetime of parts by the whole (i.e., they live and die as a whole)
  - The component classes of a composition cannot exist independent of the base class (e.g.: Window is composed of Scrollbar, Title Bar, Menu bar)

4.4 Generalization

- Represents Inheritance in which a class "gains" all of the attributes and operations of the class it inherits from, and can override/modify some of them, as well as add more attributes and operations of its own
- Inheritance is also called an "Is-A" relationship. (e.g.: Student class inherits from Person class, hence Student is a Person)
- Denoted using the hollow arrow that points from the derived class to the base class (i.e. from Student to Person)
- UML can represent multiple inheritance (one derived class inherits from multiple base classes, e.g. TA inherits from Student, Employee) even though languages like Java don't allow it

4.5 Dependency

- A dependency relationship is a semantic relationship in which one element, the client, uses or depends on one or more other elements, the supplier(s)
- Dependency is displayed as a dashed line with an open arrow that points from the client to the supplier.

5 Component Diagram

- Components are replaceable and executable pieces of a system whose implementation details are hidden
- Component diagram describes the organization of the physical and conceptual components in a system

- A **Component** is represented as a rectangle with the stereotype (the text "<<component>>") and the component name below that, as well as the component icon at the top right
- A **provided interface** is one that is provided (i.e. implemented) by the component itself. It is represented as the lollipop symbol (circle with a line at the end away from the component)
- **Required Interface** represents an interface that the component requires. It is represented as a line with a half circle at the end
- **Port** specifies a distinct interaction point or a window into an encapsulated component. (represented as a square on the edge of the component)
- Each port provides or requires one or more specific interfaces. Ports can be named, and the name is placed near the square symbol
- Connectors:
  - **Direct Connectors** link 2 components
  - An **assembly connector** links together two ports by relating together the required and the provided interfaces using ball and socket notation
  - A **delegation connector** connects a part within a component to a port of that component, thus providing/consuming services to/from the outside world of the larger component

6 Deployment Diagram

- Maps software architecture to hardware components that actually execute said software.
- Shows how the components described in component diagrams are deployed in hardware
- **Components with artifacts** demonstrate the software modules, along with the real world **artifacts** (code, libraries, exe files, config, user manuals/documentation) that participate in the execution
- **Node** is computing hardware upon which component artifacts are deployed for execution
- A **Deployment specification** is a configuration file, such as a text file or an XML document. It describes how an artifact is deployed on a node.

7 Activity Model

Refer to this link

8 Sequence Model

Refer to this link

9 State Diagram

Refer to this link