Chapter 1

Introduction to Design Principles

Course: Software Design & Architecture
Dep’t: 3rd year software engineering (R)
By Tilaye H.(MSc)
Address : B-601
IT office IV

Software Architecture, definition

- The architecture of a software system defines the system in terms of computational components and interactions among those components.

  (from Shaw and Garlan, *Software Architecture, Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.)
- **Software architecture** refers to the fundamental structures of a **software** system and the discipline of creating such structures and systems.
- Each **structure** comprises **software elements**, **relations** among them, and **properties** of both elements and relations.

Architecture vs. Design

- **Architecture** serves as a blueprint for a system. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.
- It defines a structured solution to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.
- Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product.
- These decisions comprise of :–
  - Selection of structural elements and their interfaces by which the system is composed.
  - Behavior as specified in collaborations among those elements.
  - Composition of these structural and behavioral elements into large subsystem.
  - Architectural decisions align with business objectives.
  - Architectural styles guide the organization.

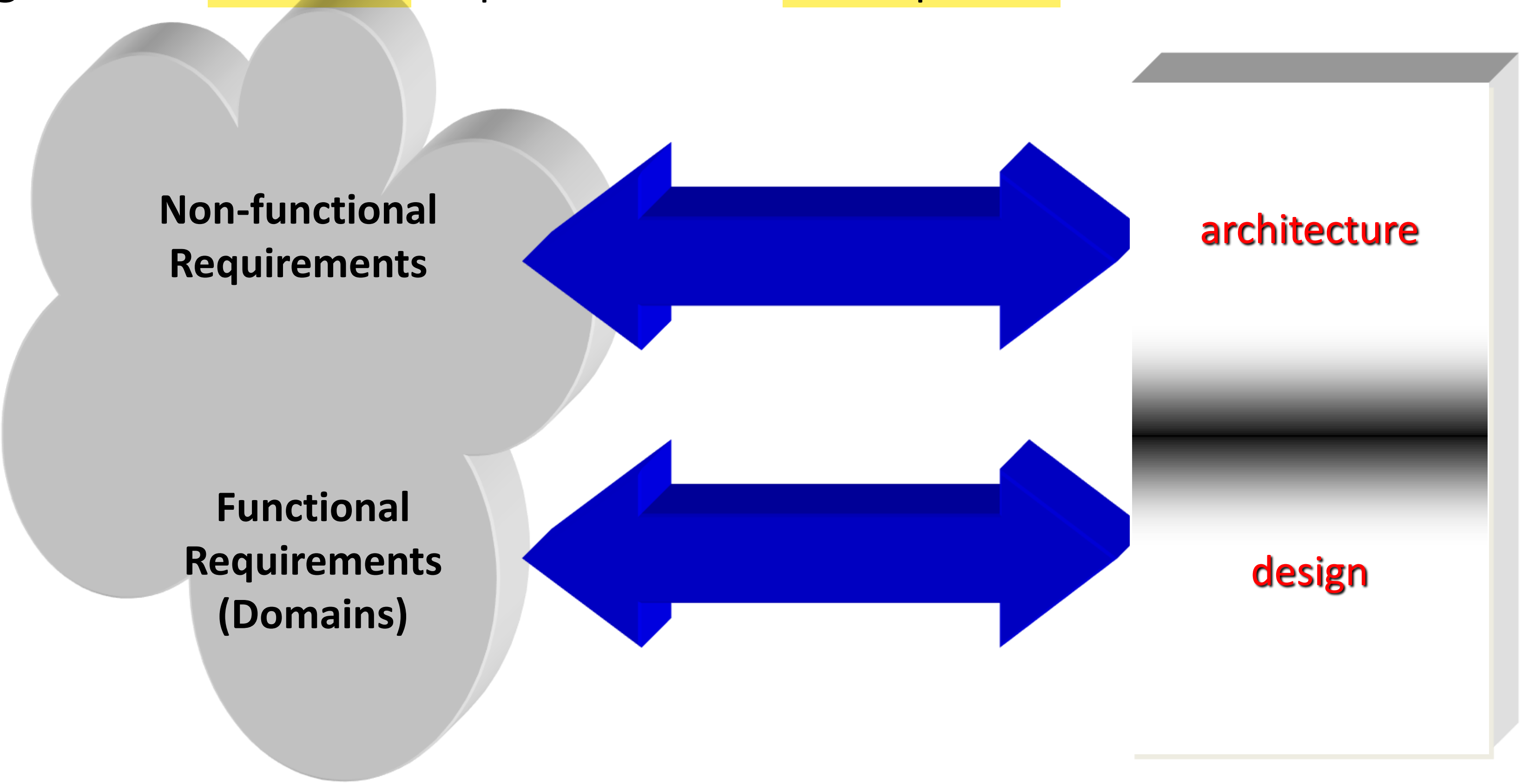
Architecture vs. Design…

- Software design provides a **design plan** that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system. The objectives of having a design plan are as follows −
- To negotiate system requirements, and to set expectations with customers, marketing, and management personnel.
- Act as a blueprint during the development process.
- Guide the implementation tasks, including detailed design, coding, integration, and testing.
- It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.

Architecture vs. Design…

Architecture: where non-functional decisions are cast, and functional requirements are partitioned

Design: where functional requirements are accomplished



Goals of software architecture

- The primary goal of the architecture is to identify requirements that affect the **structure of the application.**
- A well-laid architecture reduces the business risks associated with building a technical solution and builds a bridge between business and technical requirements.
- Some of the other goals are as follows –
  - Expose the structure of the system, but hide its implementation details.
  - Realize all the use-cases and scenarios.
  - Try to address the requirements of various stakeholders.
  - Handle both functional and quality requirements.
  - Reduce the goal of ownership and improve the organization's market position.
  - Improve quality and functionality offered by the system.
  - Improve external confidence in either the organization or system.

Software Architecture & Quality

- The notion of quality is central in software architecting: software architecture is devised to gain insight in the qualities of a system at the **earliest possible stage.**
- Some qualities are **observable via execution**: performance, security, functionality etc.
- And some are **not observable via execution**: portability, reusability, integrability, testability

Role of Software Architect

- Excellent software engineering skills
- Lead technical development team by example
- Solve the hard problems
- Understand impact of decisions
- Defend architectural design decisions
- Know and understand relevant technology
- Track technology development

Design

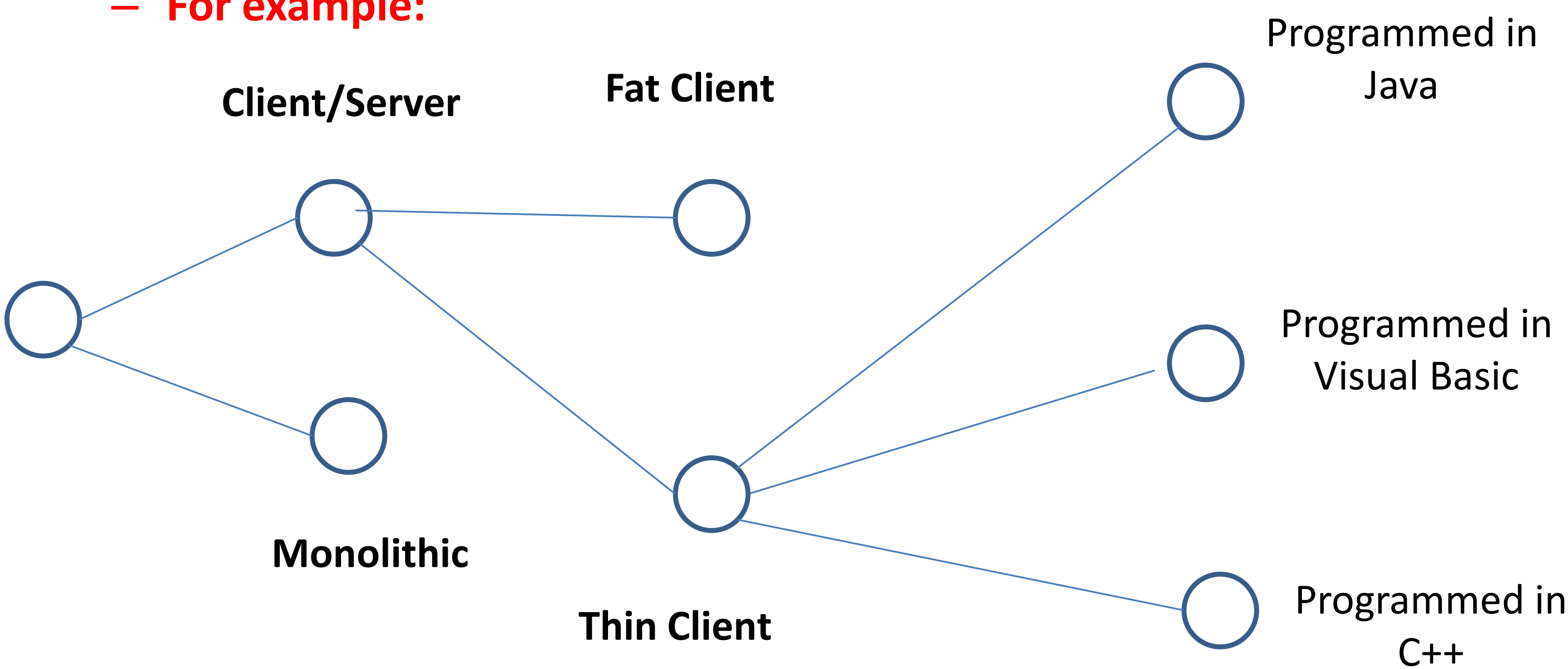
- **Definition**:
  - ***Design*** is a problem-solving process whose objective is to find and describe a way:
    - To implement the **system's *functional requirements*** while respecting the constraints imposed by the *quality and* budget."

Design as a series of decisions

- A designer is faced with a series of *design issues:*
  - Each issue normally has several alternative solutions:
    - design *options*
  - The designer makes a ***design decision*** to resolve each issue.
    - This process involves choosing the best option from among the alternatives.
- **Making decisions-** To make each design decision, the software engineer uses:
  - Knowledge of
    - the **requirements**
    - the **design** as created so far
    - the **technology** available
    - software **design principles**

Design space

- The **space of possible designs** that could be achieved by choosing different sets of alternatives is often called the *design space*
  - **For example:**



Principles Leading to Good Design

- Overall *goals* of good design:
  - Increasing profit by reducing cost and increasing revenue
  - Ensuring that we actually conform with the requirements
  - Accelerating development
  - Increasing qualities such as:
    - Changeability
    - Extensibility
    - Reusability

Changeability

- Existing requirements change and new ones are added.
- To **reduce maintenance costs** and the workload involved in changing an application, it is important to prepare its architecture for modification and evolution.
- **Two reasons why software ages:**
- Lack of movement -software ages if it is **not frequently updated**.
- Ignorant surgery -changes made by people **who do not understand the original design**, **gradually destroy the architecture**.

Extensibility

- This focuses on the extension of a software system with **new features**, as well as the **replacement of components** with **improved versions** and the **removal of unwanted** or **unnecessary features and components.**
- To achieve extensibility a software system requires loosely-coupled components.

Reusability

- It promises a reduction of both cost and development time for software systems, as well as better software quality.
- Reusability has two major aspects- software development with reuse and software development for reuse:
  - **Software development with reuse** means reusing existing components and results from **previous projects** or **commercial libraries or code components**.
  - **Software development for reuse** focuses on producing components that are **potentially reusable in future projects as part of the current software development.**

Design Principle 1: Divide and conquer

- Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things.
  - Separate people can work on each part.
  - An individual software engineer can specialize.
  - Each individual component is smaller, and therefore easier to understand.
  - Parts can be replaced or changed without having to replace or extensively change other parts.

Ways of dividing a software system

- A distributed system is divided up into clients and servers
- A system is divided up into subsystems
- A subsystem can be divided up into one or more packages
- A package is divided up into classes
- A class is divided up into methods

Design Principle 2: Increase cohesion where possible

- A subsystem or module has **high cohesion if it keeps together things that are related to each other**, and keeps out other things
  - This makes the system as a whole easier to understand and change
  - **Type of cohesion:**
    - Functional, Layer, Communicational, Utility

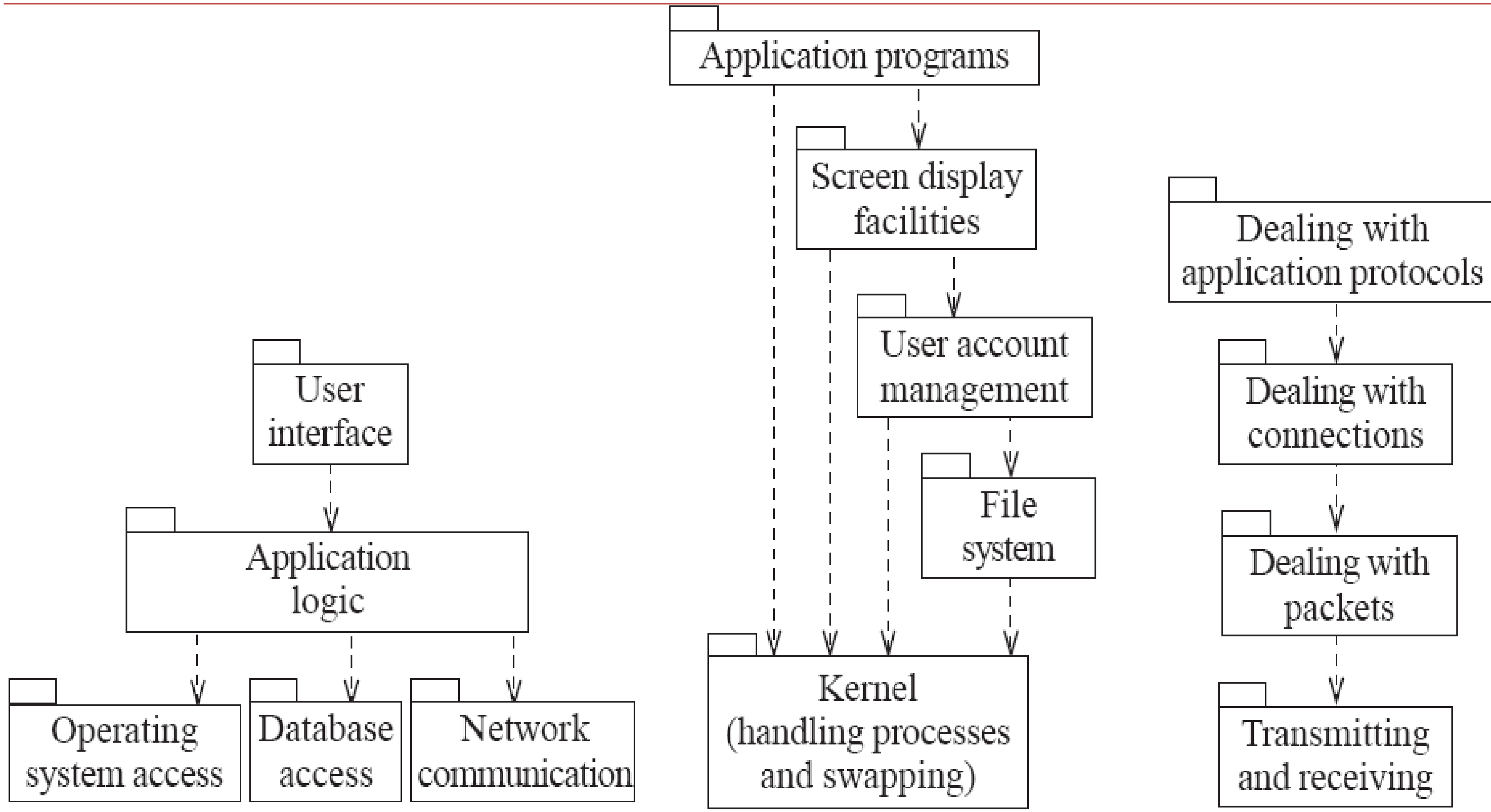
Functional cohesion

- This is achieved when ***all the code that computes a particular result* is kept together** - and everything else is kept out
  - i.e. when a module only performs a *single* computation, and returns a result, *without having side-effects*.
  - **Benefits to the system:**
    - Easier to understand
    - More reusable
    - Easier to replace
  - Modules that **update a database**, **create a new file** or **interact with the user** are **not functionally cohesive**

Layer cohesion

- **All the *facilities for providing or accessing a set of related services* are kept together, and everything else is kept out**
  - The layers should form a hierarchy
    - Higher layers can access services of lower layers,
    - Lower layers do not access higher layers
  - The set of procedures through which a layer provides its services is the *application programming interface* ***(API)***
  - You **can replace** a **layer without having any impact** on the other layers
    - You just replicate the API

Example of the use of layers



(a) Typical layers in an application program

(b) Typical layers in an operating system

(c) Simplified view of layers in a communication system

Communicational cohesion

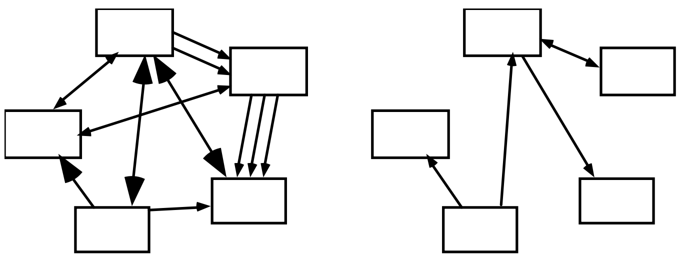
- **All the *methods that access or manipulate certain data* are kept together (e.g. in the same class)** - and everything else is kept out
  - A class would have good communicational cohesion
    - If all the system's facilities for storing and manipulating its data are contained in this class.
  - **Main advantage**: When you need to make changes to the data, you find all the code in one place
  - **Example-** StudentManager class (insert, update, delete, search etc.)

Utility cohesion

- **When *related utilities which cannot be logically placed in other cohesive units* are kept together**
  - A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable.
  - **For example**, the **`java.lang.Math`** class.

Design Principle 3: Reduce coupling where possible

- ***Coupling* occurs when there are *interdependencies* between one module and another**
  - When interdependencies exist, changes in one place will require changes somewhere else.
  - A network of interdependencies makes it hard to see at a glance how some component works.
  - **Type of coupling:**
    - Common, Control, Routine Call



Common coupling

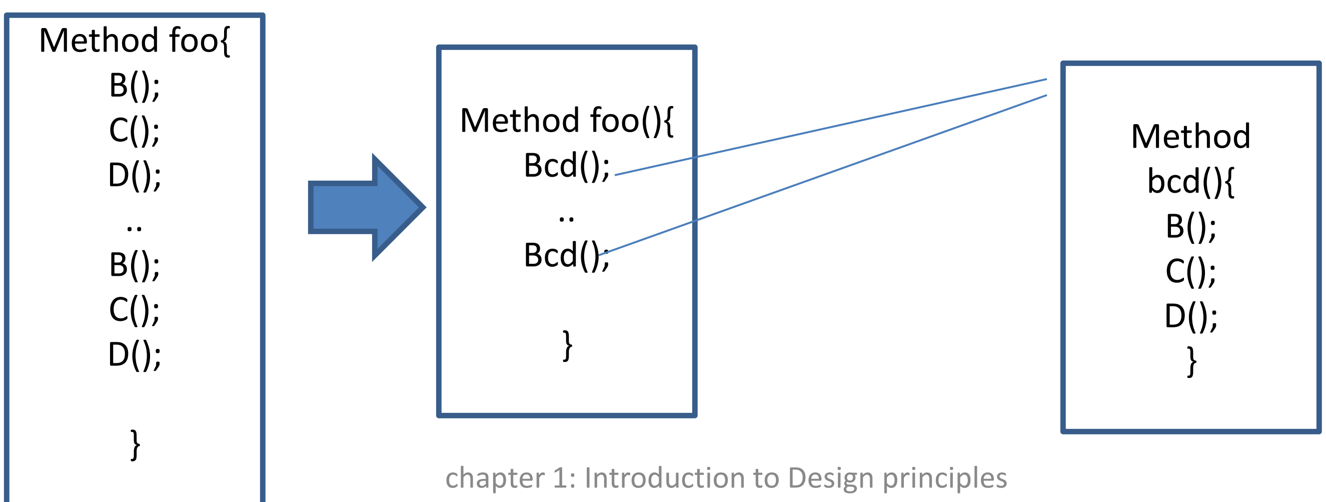
- Occurs whenever you use a *global variable*
  - All the components using the global variable become coupled to each other
  - can be acceptable for creating global variables that represent system-wide default values
  - The best way is declare all the global variables in an interface and use them through interface
  - The Singleton pattern provides encapsulated global access to an object

Control coupling

- Occurs when one procedure calls another using *a 'flag' or 'command'* that explicitly controls what the second procedure does
  - To make a change you have to change both the calling and called method
  - The use of polymorphic operations is normally the best way to avoid control coupling
  - One way to reduce the control coupling could be to have a ***look-up table***
    - commands are then mapped to a method that should be called when that command is issued

Routine call coupling

- **Occurs when one routine (or method in an object oriented system) calls another**
  - The routines are coupled because they depend on each other's behaviour
  - Routine call coupling is always present in any system.
  - If you repetitively use a sequence of two or more methods to compute something
    - then you can reduce routine call coupling by writing a single routine that encapsulates the sequence.



Design Principle 4: Keep the level of abstraction as high as possible

- Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity
  - A good abstraction is said to provide ***information hiding***
  - Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details

Abstraction and classes

- Classes are data abstractions that contain procedural abstractions
  - Abstraction is increased by defining all variables as private.
  - The fewer public methods in a class, the better the abstraction
  - abstract classes and interfaces increase the level of abstraction

Design Principle 5: Increase reusability where possible

- Design the various aspects of your system so that they can be used again in other contexts
  - Generalize your design as much as possible

Design Principle 6: Reuse existing designs and code where possible

- **Design with reuse is complementary to design for reusability**
  - Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components.

Design Principle 7: Design for flexibility

- **Actively anticipate changes** that a design may have to **undergo in the future**, and **prepare for them**
  - Reduce coupling and increase cohesion
  - Create abstractions
  - Do not hard-code anything
  - Use reusable code and make code reusable

Design Principle 8: Anticipate obsolescence

- Plan for changes in the technology or environment so the software will continue to run or can be easily changed
  - Avoid using early releases of technology
  - Avoid using software libraries that are specific to particular environments
  - Use standard languages and technologies that are supported by multiple vendors

Design Principle 9: Design for Portability

- **Have the software run on as many platforms as possible**
  - Avoid the use of facilities that are specific to one particular environment
  - E.g. a library only available in Microsoft Windows

Design Principle 10: Design for Testability

- Take steps to make testing easier
  - Design a program to automatically test the software
    - Ensure that all the functionality of the code can be driven by an external program, bypassing a graphical user interface
  - In Java, you can create a main() method in each class in order to exercise the other methods

Design Principle 11: Design defensively

- **Never trust how others will try to use a component you are designing**
  - Handle all cases where other code might attempt to use your component inappropriately means check that all of the inputs to your component are valid: **the *preconditions***
  - ***Validate the user input before using in your program***

Questions ?