# SE202
# Introduction to Software Engineering

Lecture 5-1
High level design

Pathathai Na Lumpoon

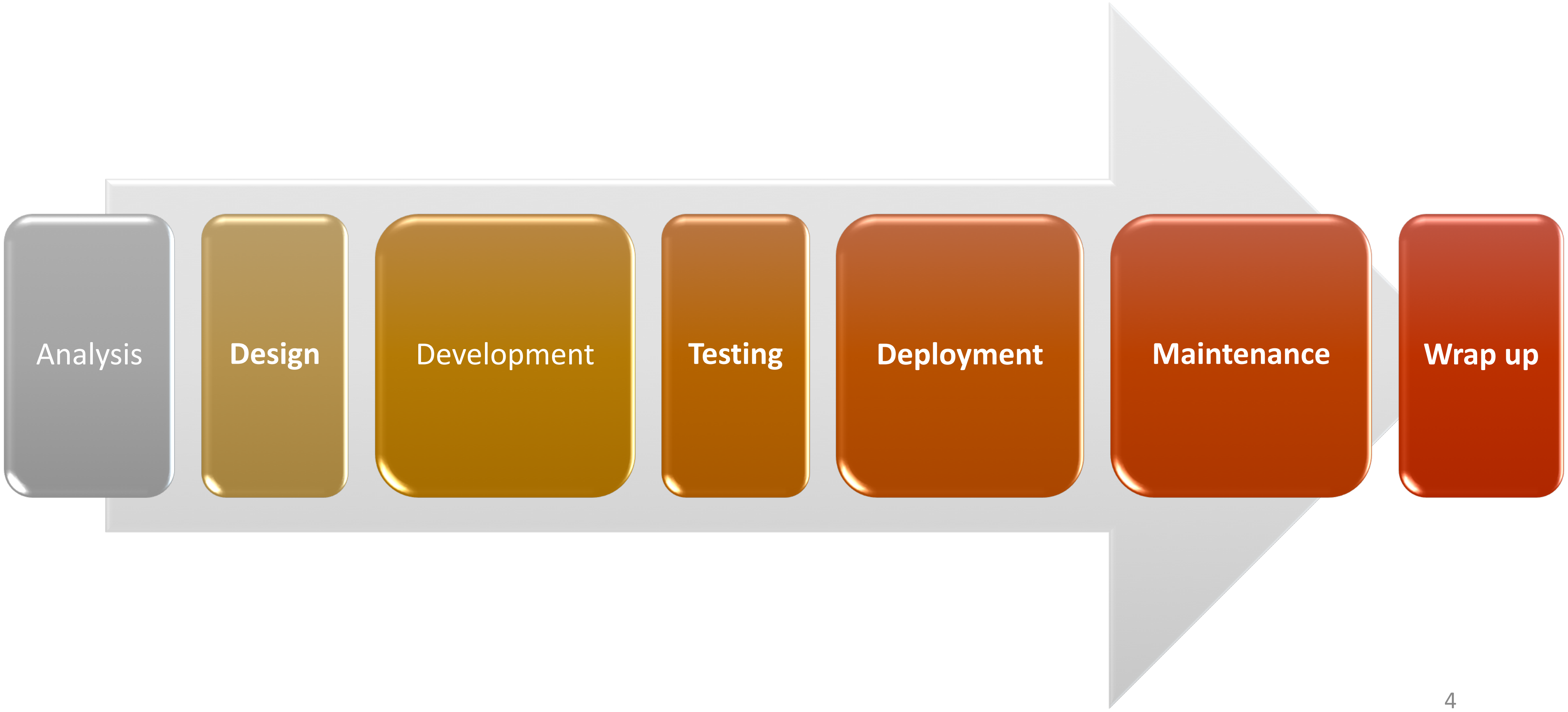
## Last class

- Use case diagram
- Requirement specification
- Requirement validation
- Change requirements

## Today

- What is design?
- High level design
    - What to specify
- Top-down versus bottom-up design
- Principles Leading to Good Design

# Software Development Life Cycle (SDLC)

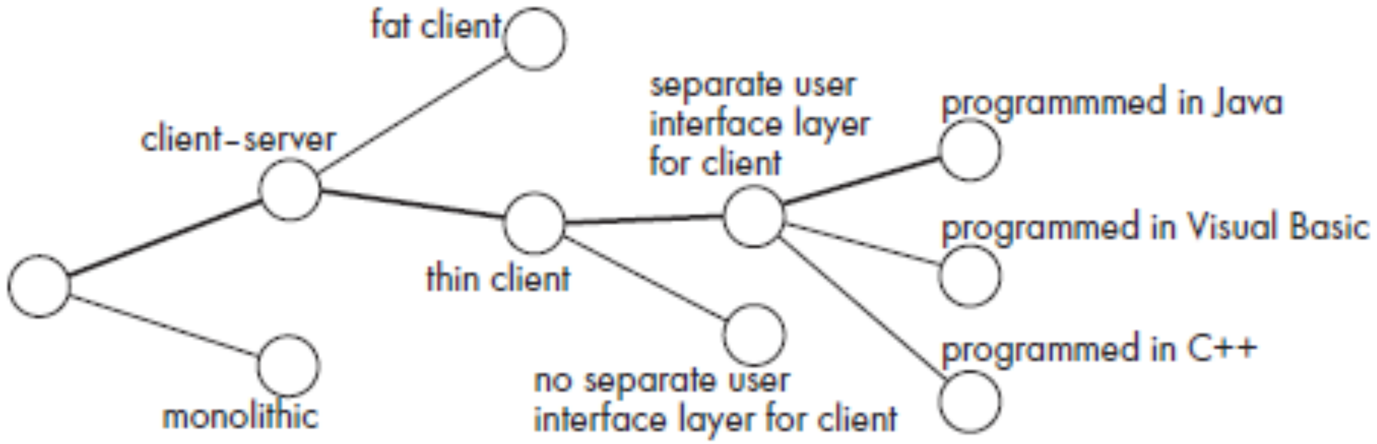


## What is design??

- **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, platform and process requirements (including the budget and deadlines), and principles of good quality.
- Knowledge the designers use
    - knowledge of the requirements;
    - knowledge of the design as created so far;
    - knowledge of the technology available;
    - knowledge of software design principles and ‘best practices’
    - knowledge about what has worked well in the past.

## Design space

- Design space is the list of possible designs that could be achieved by choosing different sets of alternatives
- E.g.


fat client
client-server
thin client
monolithic
separate user interface layer for client
no separate user interface layer for client
programmmed in Java
programmed in Visual Basic
programmed in C++

## High level design

- A view of the system at an abstract level
  - It shows how the major pieces of the finished application will fit together and interact with each other.
  - It specifies the environment in which the finished application will run.
    - Software used to develop the application
    - Hardware used to eventually run the program
- Decoupling tasks allows different teams to work on them simultaneously.
- The high-level design does not focus on the details of how the pieces of the application will work.

## WHAT TO SPECIFY (1/4)

- Security
  - Specify all the application’s security needs
  - Operating system security, Application security, Data security, Network security, Physical security
- Operating system
  - E.g. Windows, iOS, or Linux
- Hardware platform
  - E.g. desktop, laptop, tablet, phone, or mainframe
- Other hardware
  - E.g. networks, printers, programmable signs, pagers, audio, or video

# WHAT TO SPECIFY (2/4)

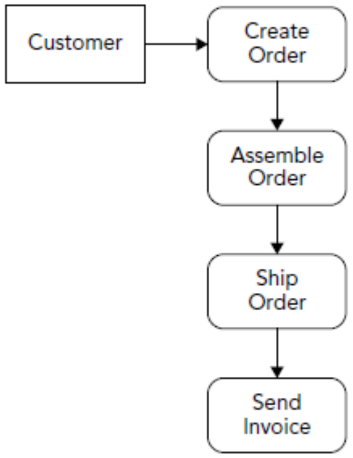
- User interface style
  - navigational techniques, menus, screens, or forms
- Internal interfaces
  - The interaction between program pieces
    - Specify data format used to exchange the data between program pieces
  - The teams assigned to the pieces can work separately without needing constant coordination.
- External interfaces
  - The interactions interact with external systems
- Algorithm
  - The design of computational mechanisms

# WHAT TO SPECIFY (3/4)

- Architecture
  - E.g. monolithic, client-server, multitier, component-based, service-oriented, data-centric, event driven, rule-based, or distributed and etc.
- Reports
  - E.g. application usage, customer purchases, inventory, work schedules
- Database
  - E.g. database platform, major tables and their relationships, user access, maintenance, backup

## WHAT TO SPECIFY (4/4)

- Top-level classes
  - E.g. Customer , Employee , and Order
- Data Flows
  - Flows of data among different processes
  - E.g A data flow diagram shows how data such as a customer order flows through various processes.



# Top-down and bottom-up design

- Top-down design
    - First design the very high level structure of the system.
    - Then gradually work down to detailed decisions about low-level constructs.
    - Finally arrive at detailed decisions such as:
        - the format of particular data items;
        - the individual algorithms that will be used.

## Top-down and bottom-up design

- Bottom-up design
  - Make decisions about reusable low-level utilities.
  - Then decide how these will be put together to create high-level constructs.

- A mix of top-down and bottom-up approaches are normally used:
  - Top-down design is almost always needed to give the system a good structure.
  - Bottom-up design is normally useful so that reusable components can be created.

## 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
  - Usability
  - Efficiency
  - Reliability
  - Maintainability
  - Reusability

## Principles Leading to Good Design

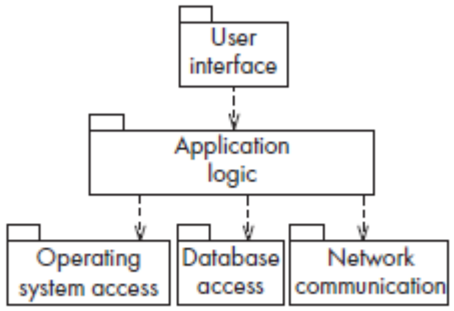
1. Divide and conquer
2. Increase cohesion where possible
3. Reduce coupling where possible
4. Keep the level of abstraction as high as possible
5. Increase reusability where possible
6. Design for flexibility
7. Anticipate obsolescence
8. Design for portability
9. Design for testability
10. Design defensively

## Principle 1: Divide and conquer

- Dividing things up into smaller chunks to achieve the goal.
- E.g.
  - A car is divided into smaller, more manageable chunks – each assembly line worker will focus on one small task.
  - 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.
- Benefits
  - Separate people can work on each part.
  - Each individual component is easier to understand.
  - Opportunities arise for making the components reusable.

# Principle 2: Increase cohesion where possible

- Cohesion is a way to keep things together that belong together.
- An entity keeps together things that are related to each other and keeps out other entities. Examples:
  - A module that computes a mathematical function sun as since or cosine.
  - Higher level layers provide a set of related services to users.

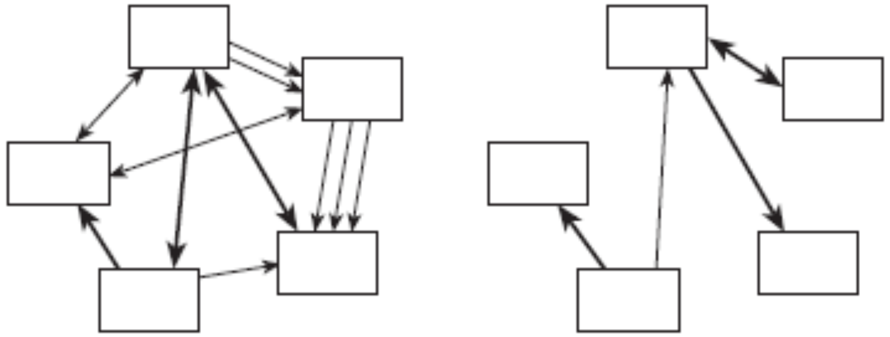


(a) Typical layers in an application program

Benefits:
It is easier to understand a module.
Increase reusable

## Principle 3: Reduce coupling where possible

- Coupling occurs when there are interdependencies between one module and another.
- The more tightly coupled a set of modules is, the harder it is to understand and, hence, change the system.
- To reduce coupling, you have to reduce the *number* of connections between modules and the *strength* of the connections.



**Abstract examples of a tightly coupled system (left) and a loosely coupled system (right). The boldness of the arrows indicates the strength of the coupling**

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

- Abstraction is a way to hide the details to reduce the complexity
- Abstractions work by allowing you to understand the essence of something and make important decisions without knowing unnecessary details.
- Ex. when creating class diagrams, you often initially leave out the data types of attributes, and you do not show the implementation details of associations.

## Principle 5: Increase reusability where possible

- Two approaches: design *for* reuse and design *with* reuse.
- Generalize a design as much as possible.
  - Several other systems can use reusable component.
  - E.g. if you are creating a facility to draw a particular kind of diagram, why not design it so that it could be used to draw other kinds of diagrams for other applications?

## Principle 6: Design for flexibility

- Designing for *flexibility* (also known as *adaptability*) means actively anticipating changes that a design may have to undergo in the future and preparing for them.
- Such changes might include changes in implementation or changes in functional requirements.

## Principle 7: Anticipate obsolescence

- Anticipating obsolescence means planning for evolution of the technology or environment so that the software will continue to run or can be easily changed

## Principle 8: Design for portability

- The ability to have the software run on as many platforms as possible

## Principle 9: Design for testability

- The most important way to design for testability is to ensure that all the functionality of the code can be executed with the various inputs.

## Principle 10: Design defensively

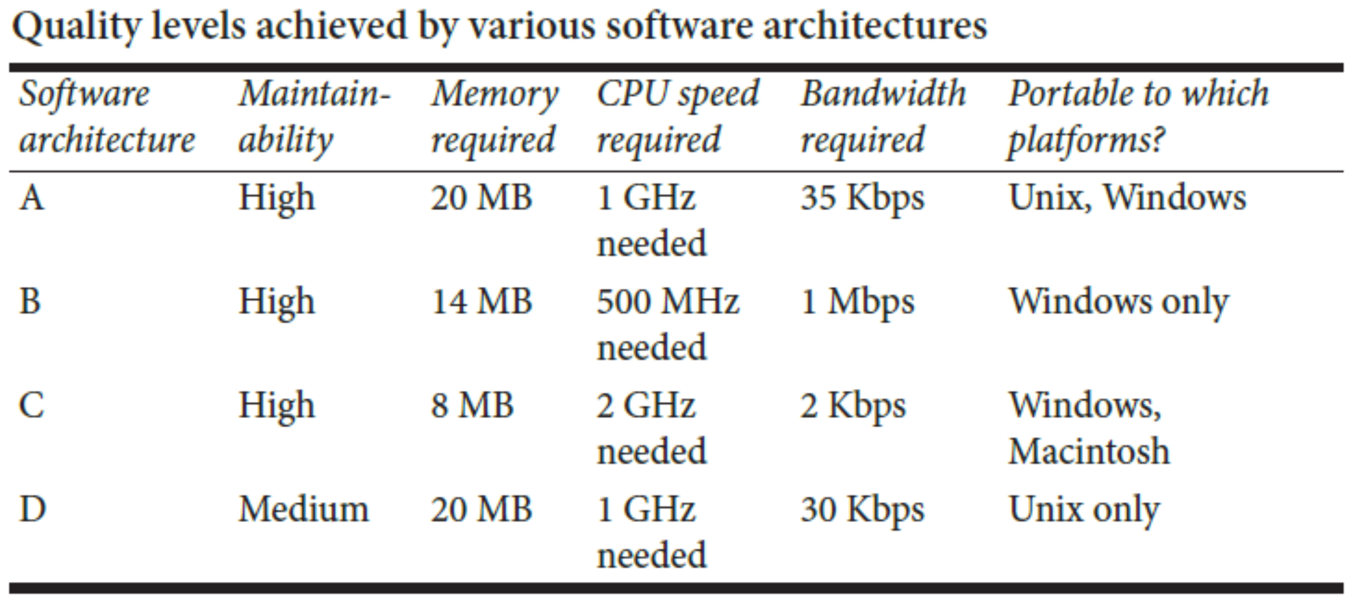
- The most important way to design defensively is to check that ALL of the inputs to your component are valid.

## Techniques for making good design decisions

- Using objectives and priorities to decide among alternatives
  - An objective is a measurable value you wish to attain.
  - A priority states which qualities override others in those cases where you must make compromises.
- The qualities to consider when setting priorities and objectives include memory efficiency, CPU efficiency, maintainability, portability and usability.

# Example

- Give the table showing the quality levels achieved by various software architectures.



**Quality levels achieved by various software architectures**

| *Software architecture* | *Maintain-ability* | *Memory required* | *CPU speed required* | *Bandwidth required* | *Portable to which platforms?* |
|---|---|---|---|---|---|
| A | High | 20 MB | 1 GHz needed | 35 Kbps | Unix, Windows |
| B | High | 14 MB | 500 MHz needed | 1 Mbps | Windows only |
| C | High | 8 MB | 2 GHz needed | 2 Kbps | Windows, Macintosh |
| D | Medium | 20 MB | 1 GHz needed | 30 Kbps | Unix only |

- Determine which architecture you might choose if you had the following objectives and priorities.
    - Objectives: runs on Windows; works on a 30 Kbps connection or faster; works on a 1 GHz machine or faster; requires no more than 25 MB memory.
    - General priorities, starting with first: bandwidth efficiency, CPU efficiency, portability, memory efficiency, maintainability.