Software Engineering

National Chiao Tung University
Chun-Jen Tsai
05/09/2012

Complications of Software Design

- Software bugs have caused large scale disasters
- Software engineering → Try to find a better way to develop and maintain a reliable software system
- Software Engineering is different from other engineering disciplines:

	Traditional Engineering	Software Engineering
"Off-the-shelf" parts available	Often	Sometimes
Required Performance	Within tolerance	Perfect
Quality Metrics	Mean time between Failure (MTBF)	Unclear
Scientific Basis	Physics	Unclear

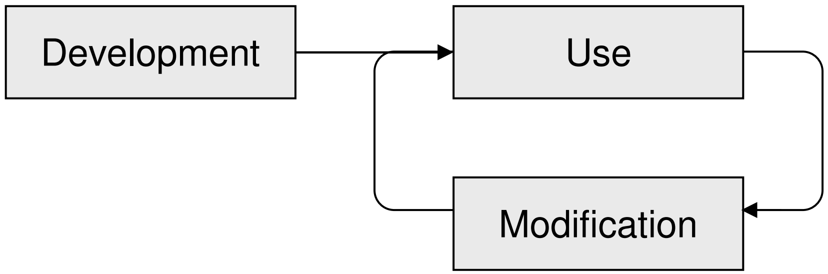
CASE Tools

- An important topic in software engineering is the design of Computer-Aided Software Engineering (CASE) tools for:
  - Project planning – for resource (time, personnel) allocation
  - Project management – for status tracking
  - Documentation – for semi-automatic document generation†
  - Prototyping and simulation – for fast proof-of-concept
  - Interface design – for GUI development
  - Programming – for program coding, version control, and debugging

† Please google "Doxygen"

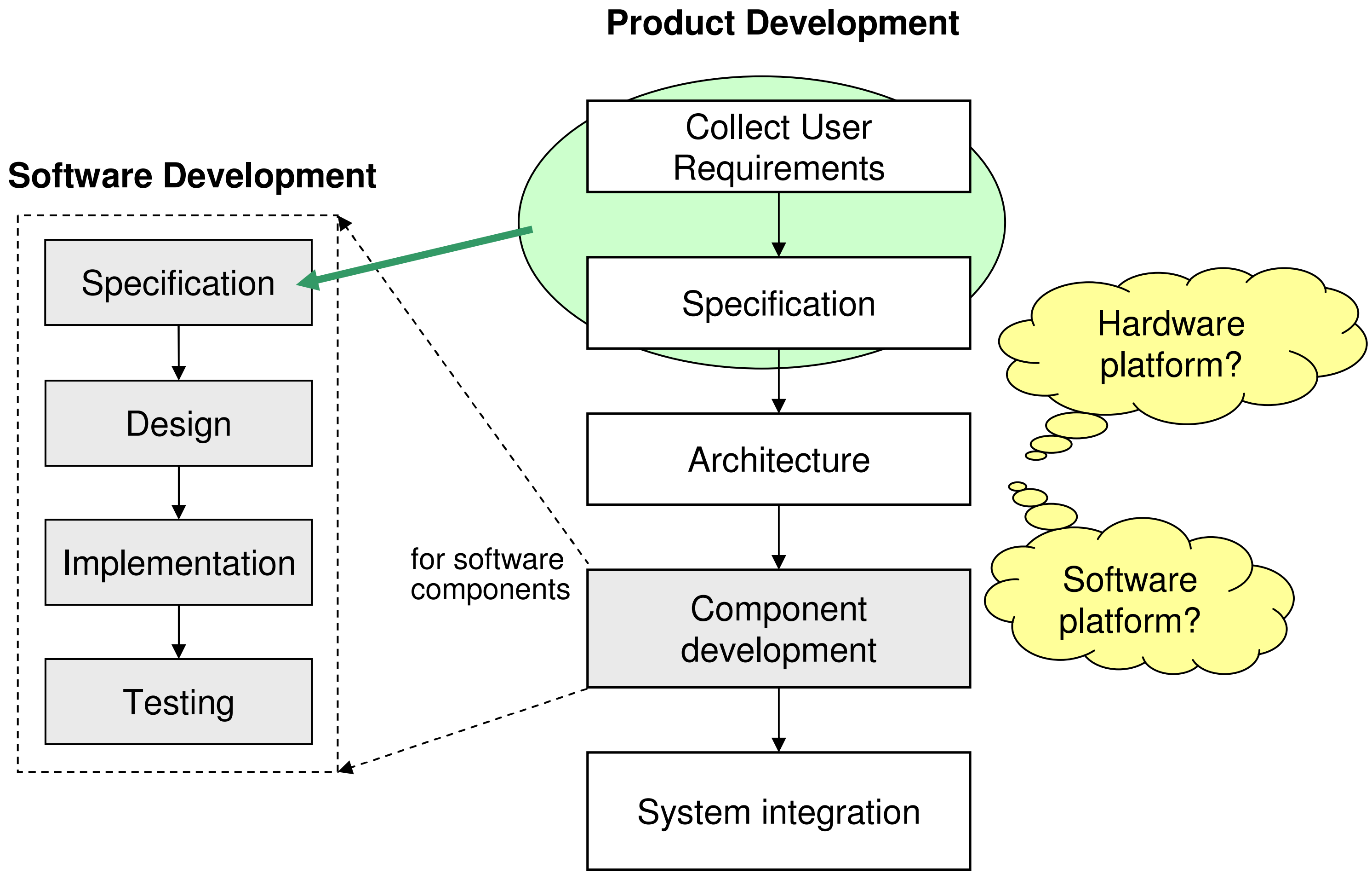
The Software Life Cycle

- The software life cycle is composed of three stages:



- For other products, the “modification” phase is called maintenance (replacing old ware-out parts); in software engineering, it’s called updates and upgrades

Software Development Phases



Specification and Design Stages

- Requirement Specification:
  - Based on *application user requirements* and some *technical specifications, perform a feasibility study*
  - The output of the analysis stage is the "*software requirement specification document*"
- Design:
  - The specification stage concentrates on ***what*** the system should do, while the design stage concentrates on ***how*** the system will accomplish those goals
  - User interface design requires mixed knowledge of arts, psychology, ergonomics, and programming

Implementation Stage

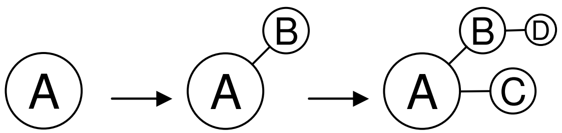
- Implementation stage creates system from design
  - Write programs
  - Create data files
  - Develop databases
- In principle:
  - A software analyst (software architect) is involved in software development at the specification and design stage
  - A programmer write programs that implement the design
- In practice, software architects and programmers are interchangeable terms

Testing Stage

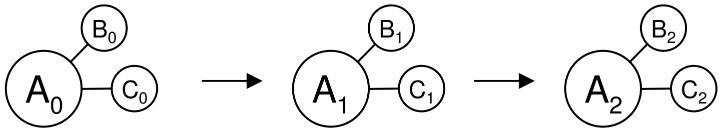
- Testing occurs in two forms
  - Validation testing – checking to see if the system meets the original requirements and specifications
  - Defect testing – identifying and correcting errors (bugs) of the system
- Granularity of testing:
  - Module testing – test a single module
    - To test a module in a system, simplified versions of other modules (called stubs) in the system are often used
  - Unit testing – test a smallest software module
    - For imperative programming language, this is often a function
    - Test codes are written to call the function with input at boundary and/or singular points
  - System testing – test the entire system

Software Development Models (1/2)

- Waterfall model
  - Strictly following the orders of the four stages of software development to design a complete system from scratch
  - Too slow to react to a dynamic environment
- Incremental model
  - Start with a simplified system, and incrementally build more-and-more complete systems



- Iterative model
  - Start with a full system with simplified modules, and incrementally build more-and-more complete modules



Software Development Models (2/2)

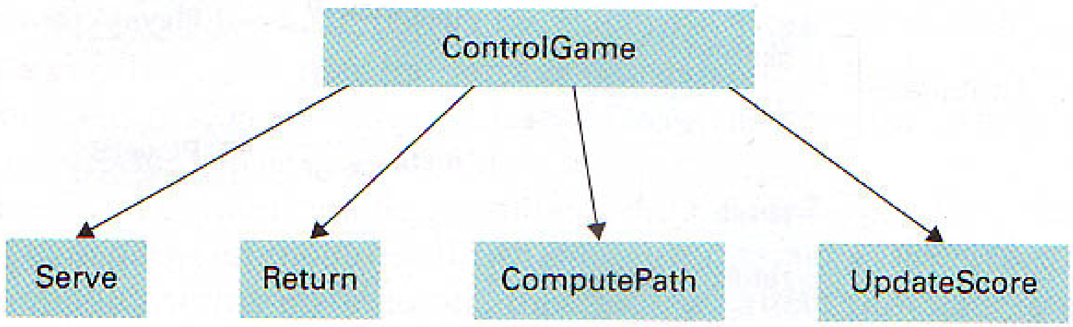
- For incremental and iterative development models, the early versions of the final system is often called prototypes
  - Evolutionary prototyping – the prototypes will be refined incrementally and eventually becomes the final system
  - Throwaway prototyping – the prototypes are only used for communication and quick verification of the design; later, a fresh implementation will be used for the real product
    - For example, rapid prototyping for UI design
- Open-source development model
- Extreme programming (XP)
  - A programming project is accomplished by a team of equal programmers cooperate in a flexible manners with repeated daily cycles of analyzing, designing, implementing, and testing

Modularity

- Software modularity can be realized in the forms of procedure, objects, and components
  - Procedures – imperative paradigm
    - Key info: procedure relations
  - Objects – object-oriented paradigm
    - Key info: object instances and object collaborations
  - Components – reusable software units, often for object-oriented paradigm
    - Key info: component interfaces
- The goal of modular design is to minimize coupling and maximize cohesion
  - Coupling: interactions between modules
  - Cohesion: internal binding within a module

Program Visualization Techniques

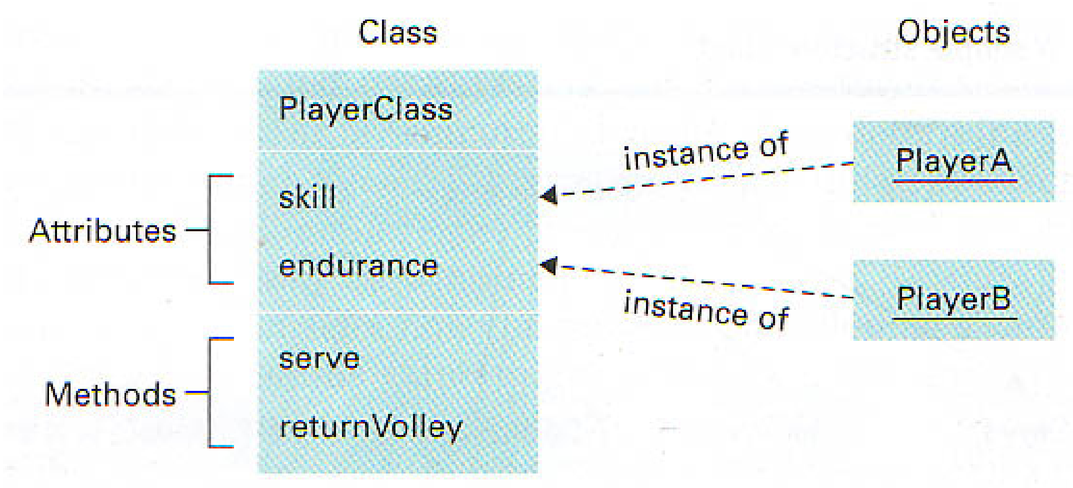
- It would be nice if we can use some diagrams to describe the relations and interactions among software modules
- Structure chart – displays relations among modules of a procedural design:



Object-Oriented Visualization (1/2)

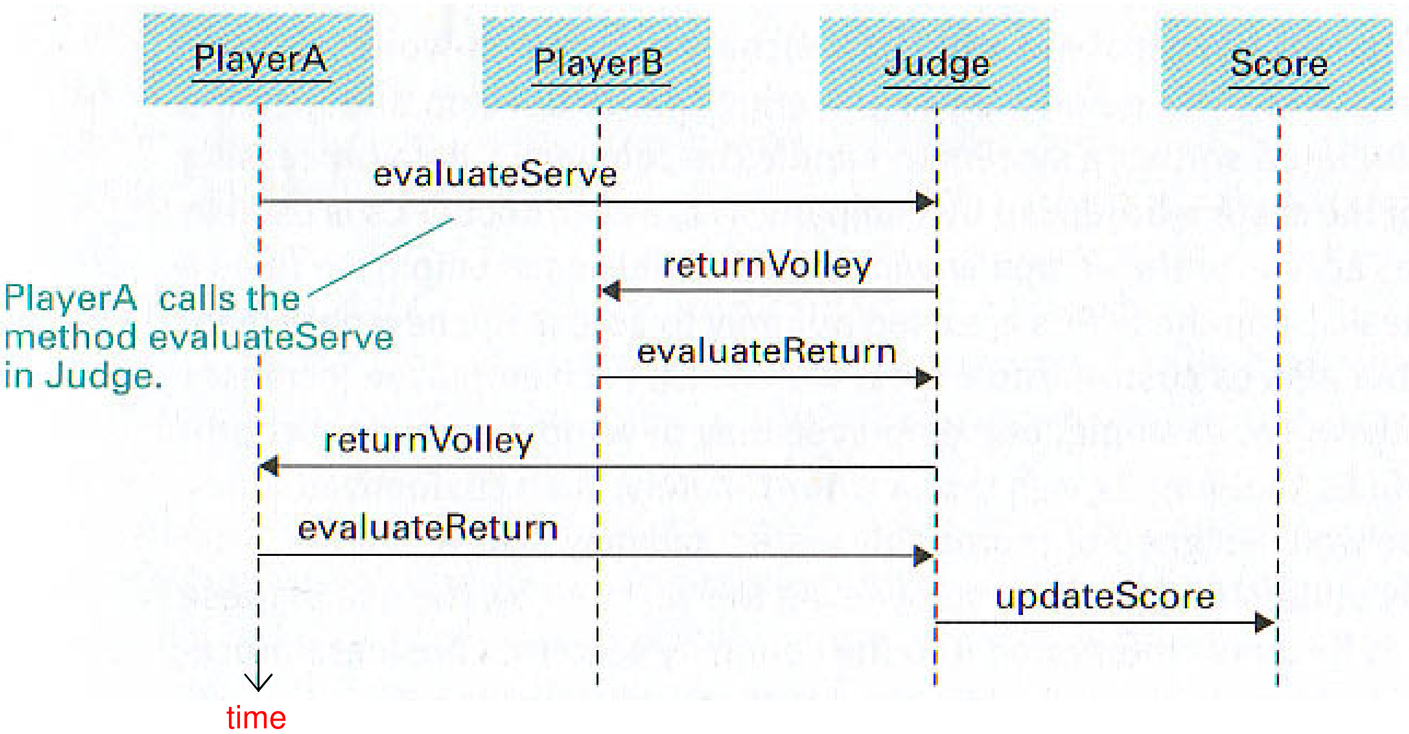
- In object-oriented designs, it takes more than structure charts to describe the relations and interactions among modules

- For object relations



Object-Oriented Visualization (2/2)

- For object interactions

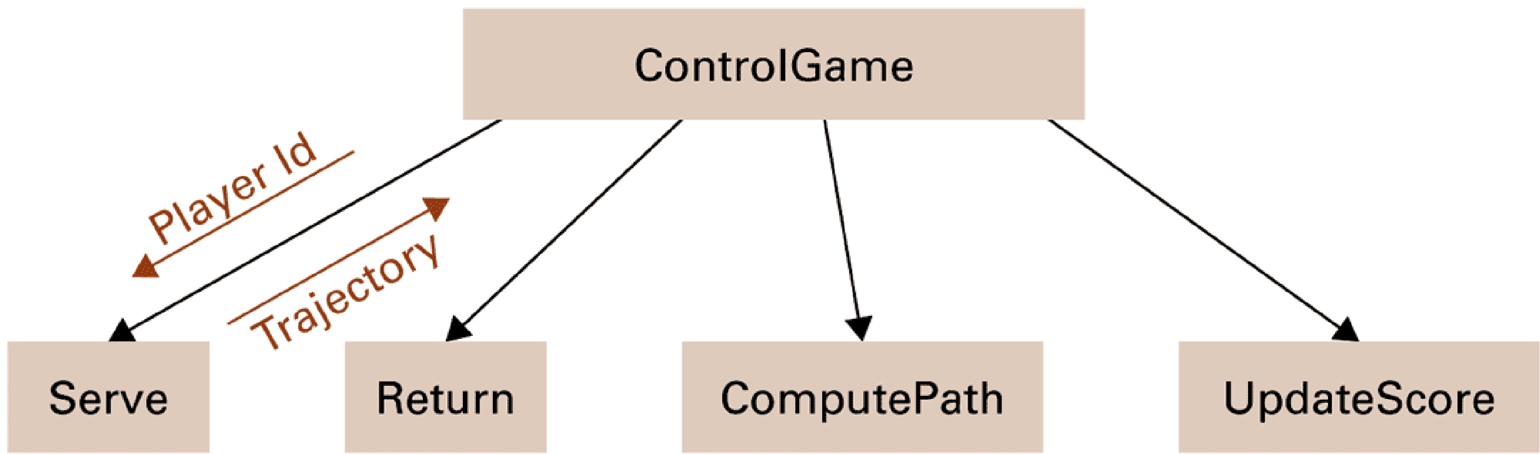


Coupling

- Control coupling
  - One module passes control to another
- Data coupling
  - Sharing of data between modules
- Implicit coupling: hidden coupling may cause errors
  - Global data: data accessible to all modules
  - Side effects: action performed by a procedure that is not readily apparent to its caller

Structure Chart with Data Coupling

- A structure chart can also shows data coupling

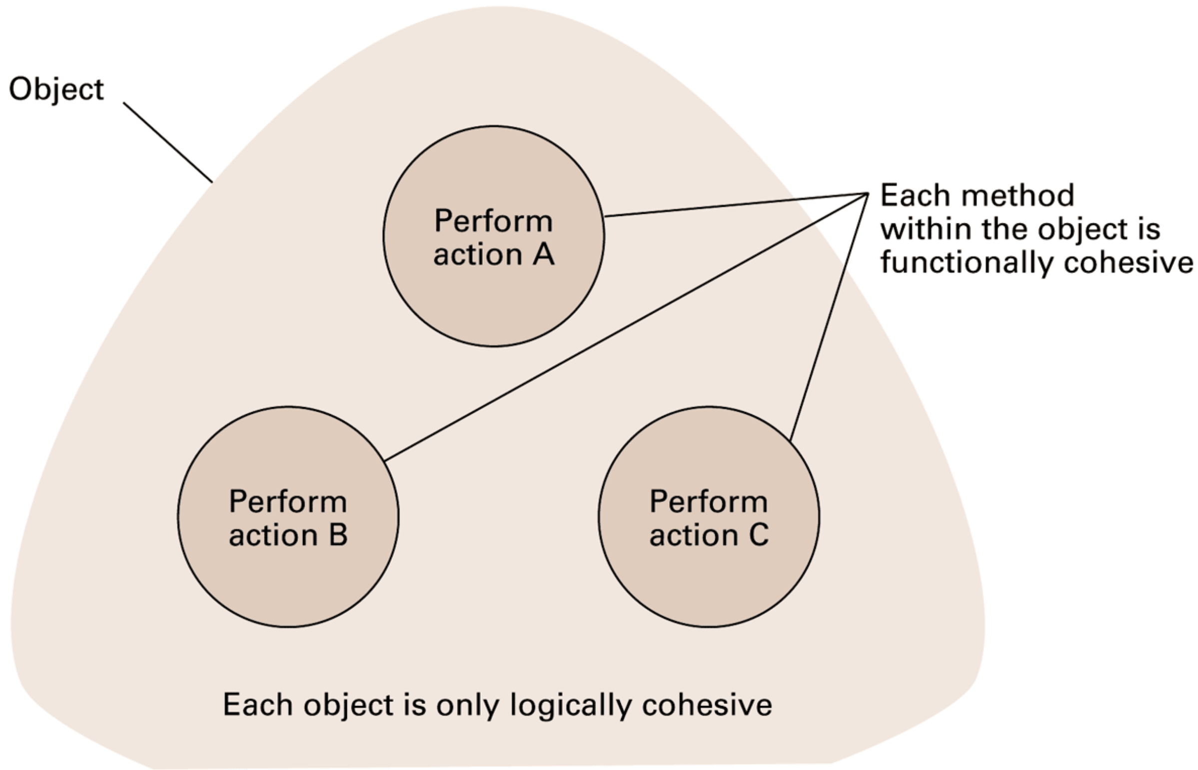


Cohesion

- Logical cohesion
  - Logical similarity of actions and components in a module
- Functional cohesion
  - Each component focus on performing a single activity
  - Stronger than logical cohesion
- Cohesion in object oriented systems
  - Entire object (i.e. the collection of data fields and methods in an object) should be logically cohesive
  - Each method (i.e. the tasks you perform in a method) should be functionally cohesive

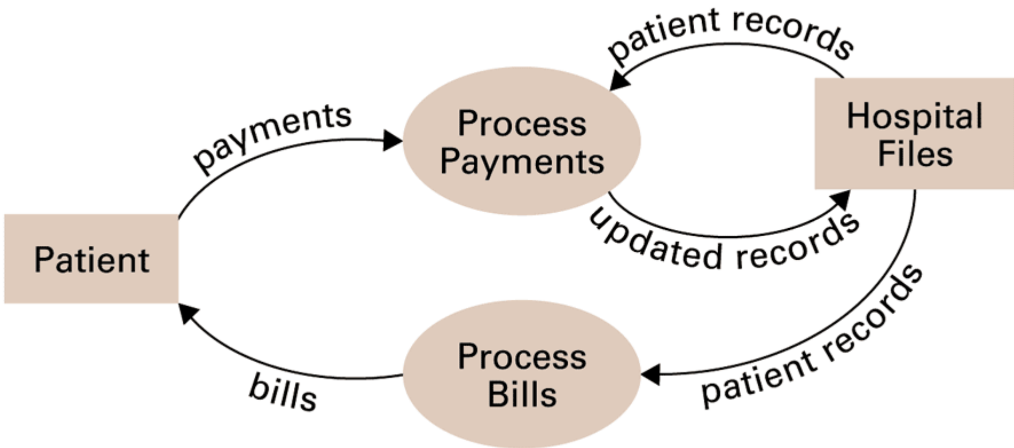
Example of Cohesion

- Logical and functional cohesion within an object:



Data-based Design Techniques

- Dataflow diagram
  - Displays how data moves through a system



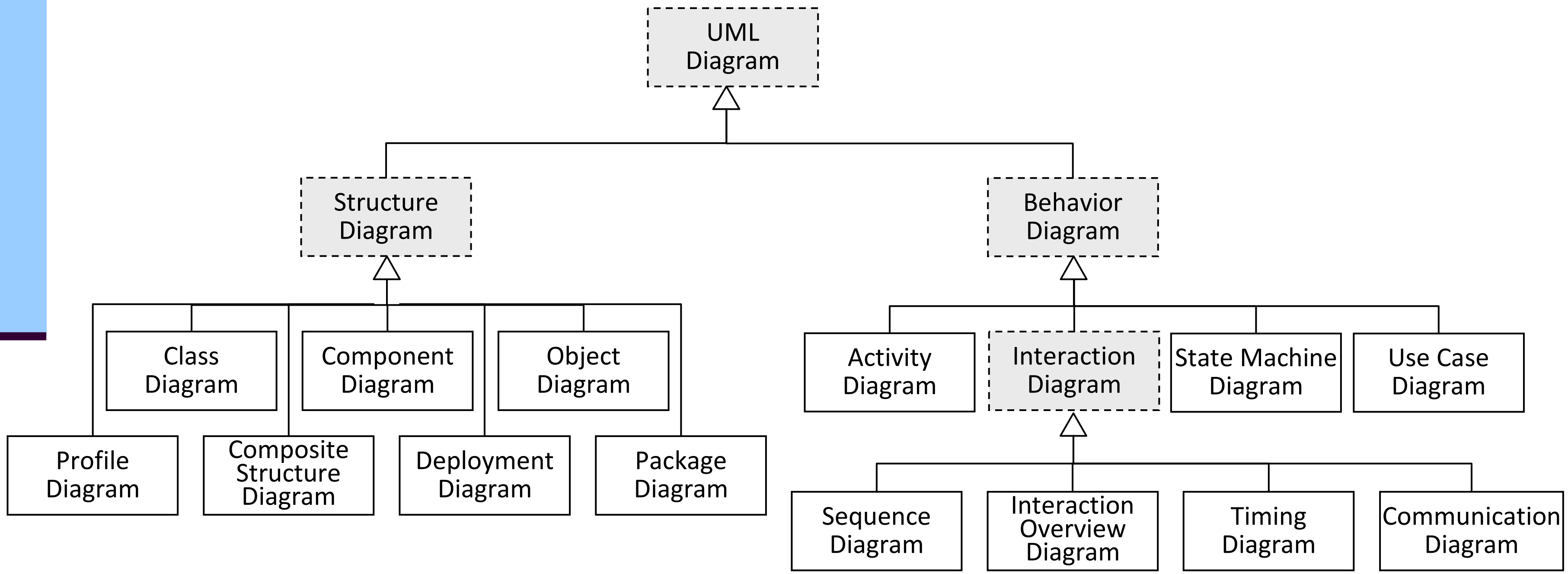
- Data dictionary
  - Central repository of information about the data items appearing throughout a software system

UML: Unified Modeling Language

- The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting large complex systems
- The development of UML began in late 1994 by Grady Booch and Jim Rumbaugh of Rational Software Corporation; Later, Ivar Jacobson of Objectory and other companies joined the effort
- UML 1.0 and later 1.1 was officially released in 1997. UML 2.0 is adopted in 2005.

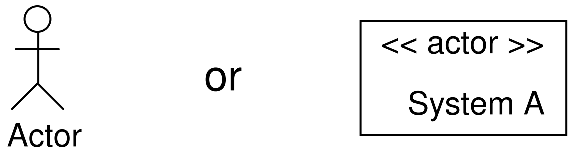
UML Diagrams

- There are 14 types of UML 2.0 diagrams:
  - Gray boxes are diagram classes;
    white boxes are instances of diagrams

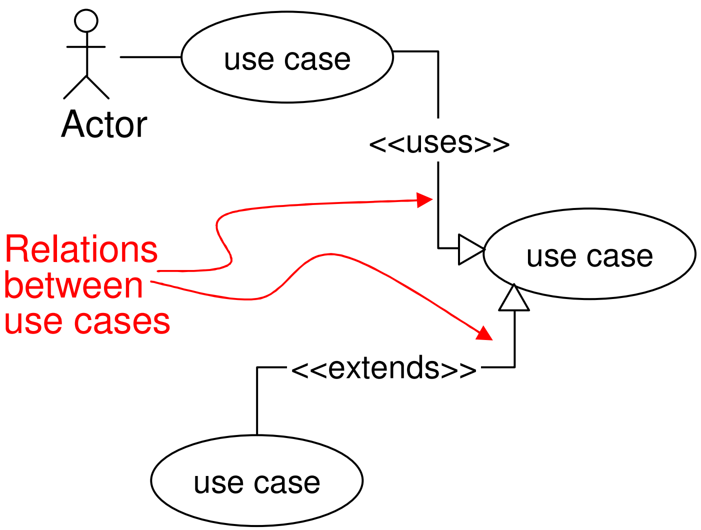


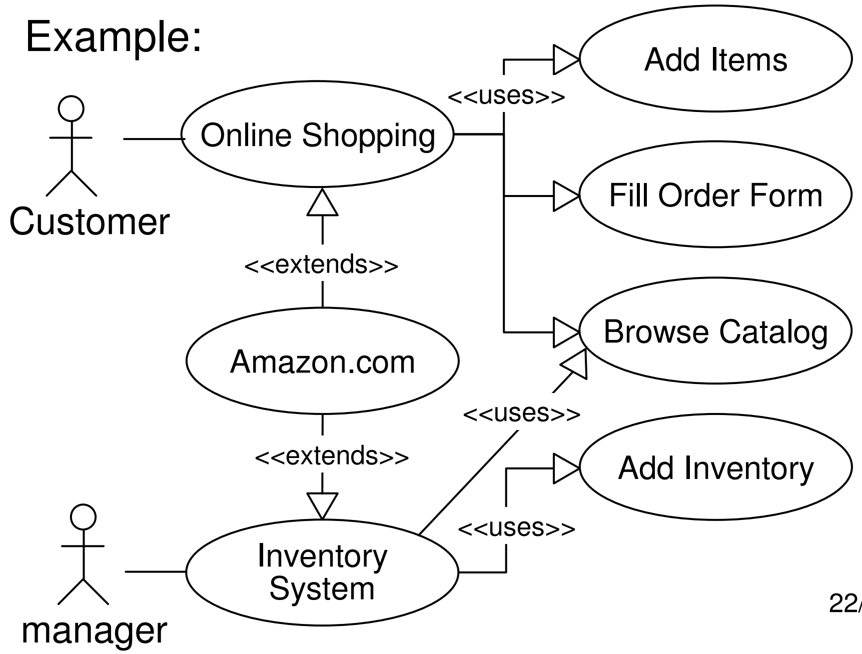
Use Case Diagram

- Actors in UML are the users of a system:

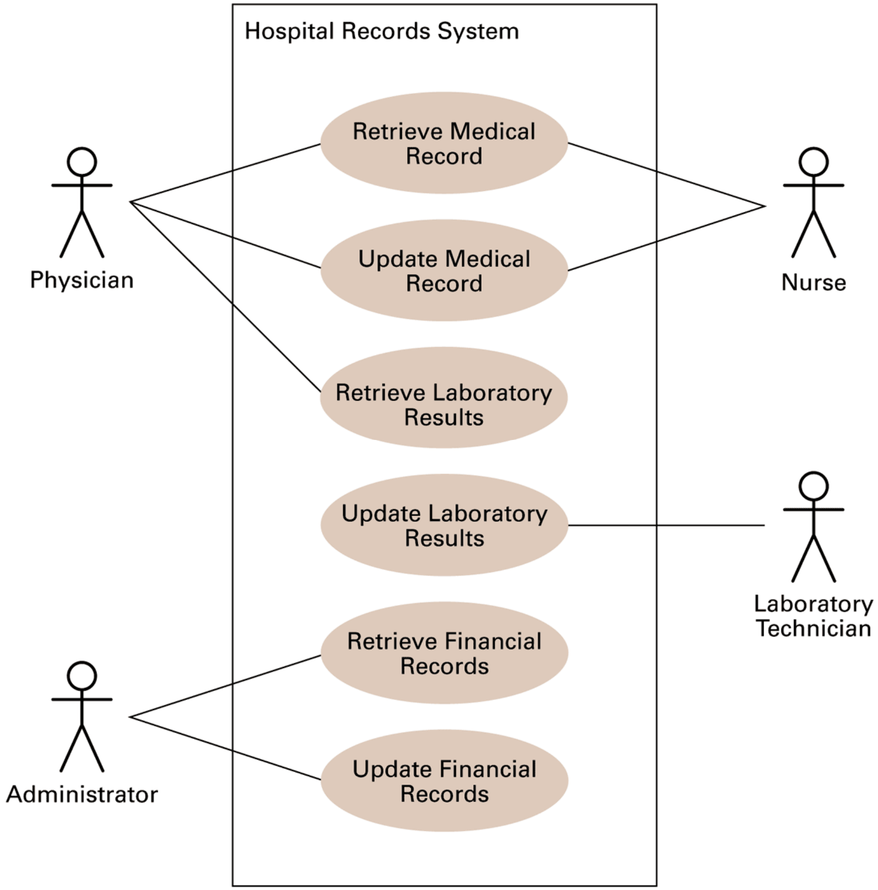


- Use case diagrams model the functions of systems



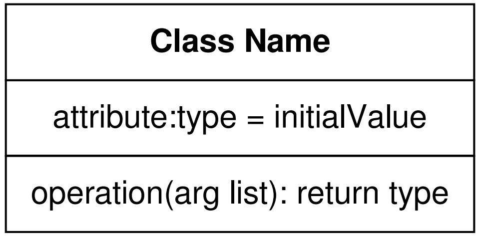


Another Use Case Example

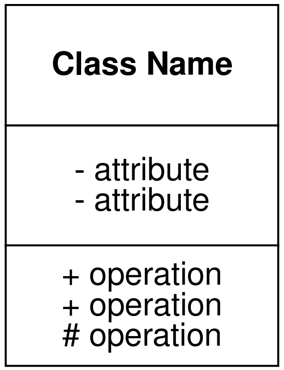


Class Diagram (1/3)

- Class diagrams describe the relations (dependencies or collaborations) among different classes
- Class definition:

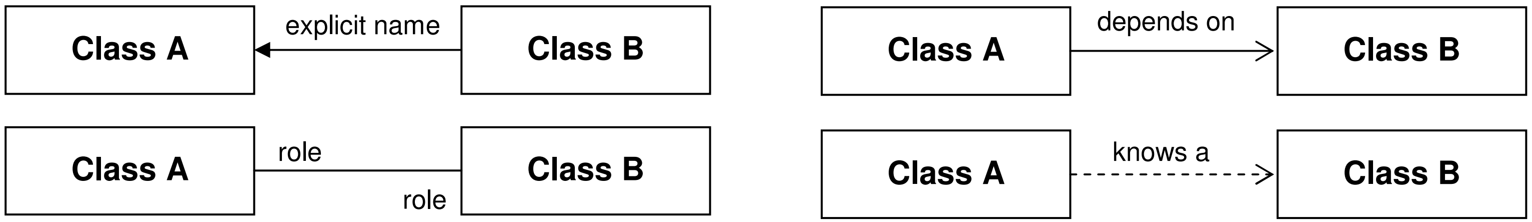


- Visibility
  - + means public
  - – means private
  - # means protected

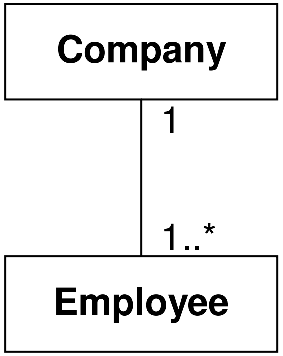


Class Diagram (2/3)

- Association

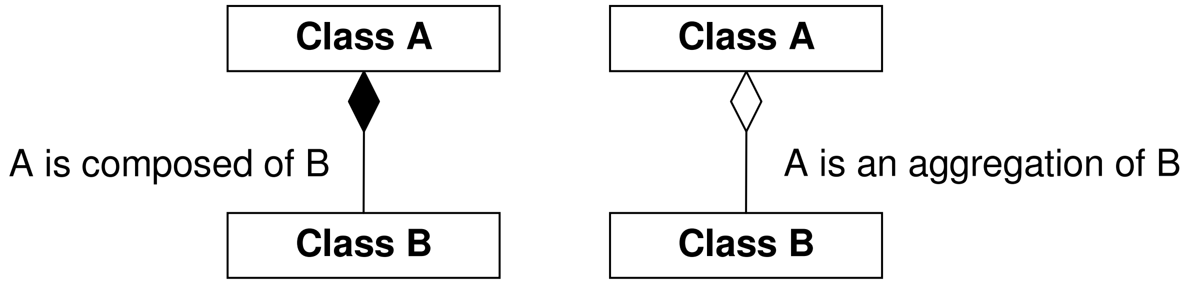


- Multiplicity
  - 1 means no more than one
  - 0..1 means zero or one
  - * means many
  - 0..* means zero or many
  - 1..* means one or many

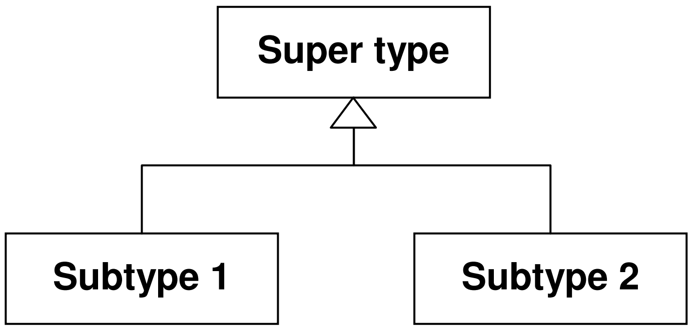


Class Diagram (3/3)

- Composition and aggregation

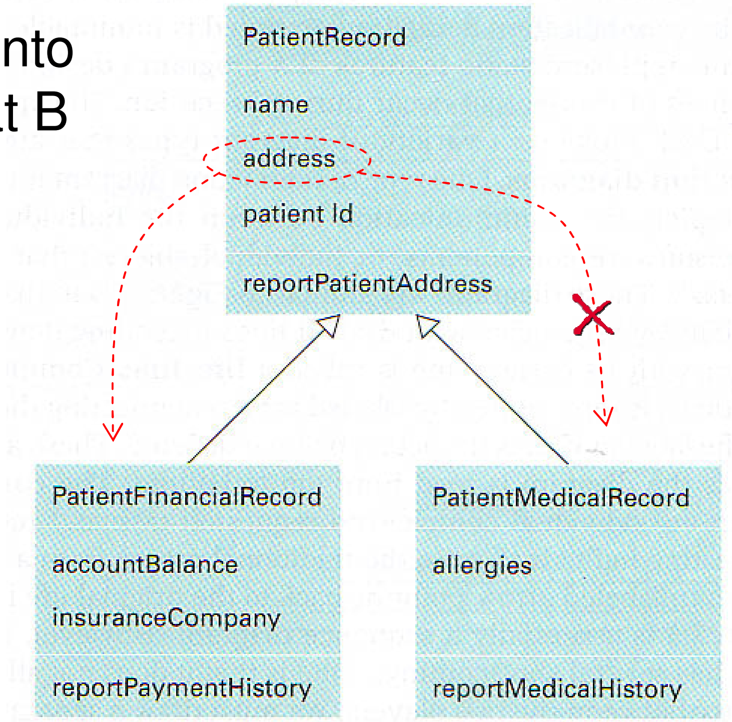


- Generalization (inheritance)



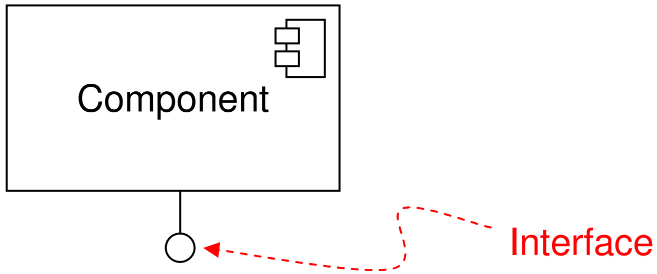
Example: Class Generalizations

- The concept of "generalization" in UML and "inheritance" in O-O languages are different
  - Inheritance has stronger binding between the parent class and the derived class
  - Generalization of class A into class B does not imply that B contains all attributes of A

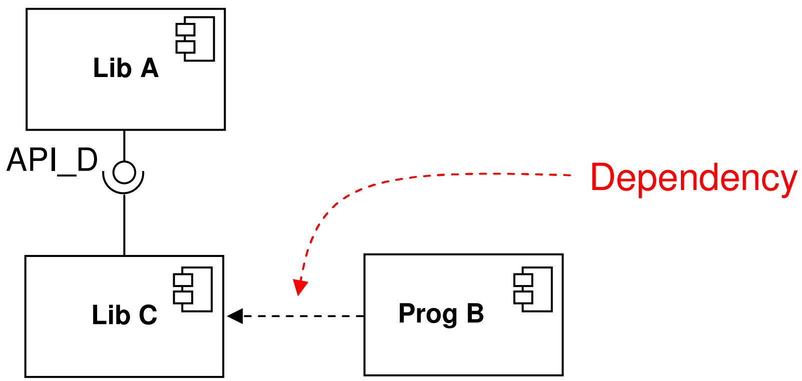


Component Diagram

- ❑ A component is a building block of the system



- ❑ Dependency of components
  - Example: Program B depends on library C which uses library A through interface API_D can be expressed as follows,

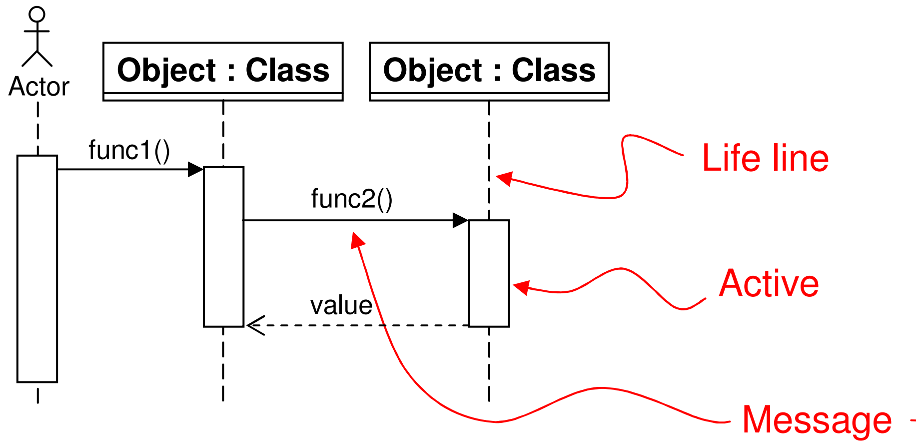


Sequence Diagram (1/2)

- Sequence diagrams present the flow of messages between instances, objects, or processes along time
- The time-axis of a sequence diagram progresses in the vertical direction
- Messages flow can be synchronous (blocking) or asynchronous (non-blocking)

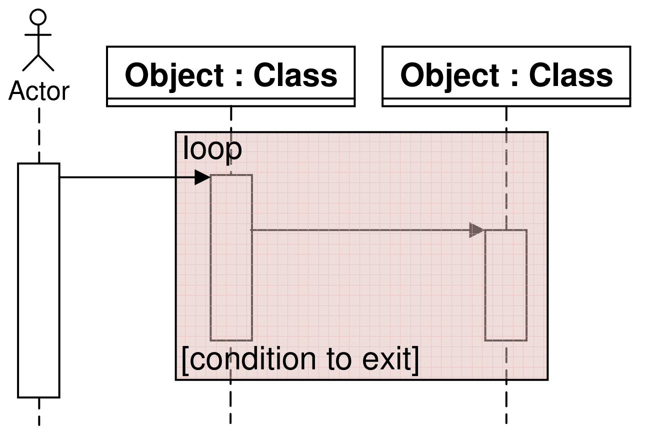
Sequence Diagram (2/2)

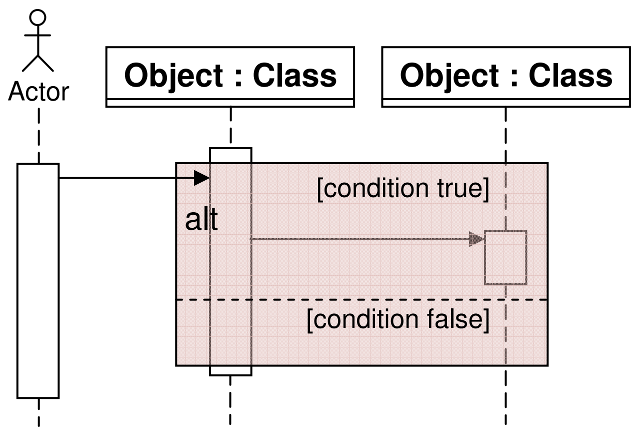
- Call sequence



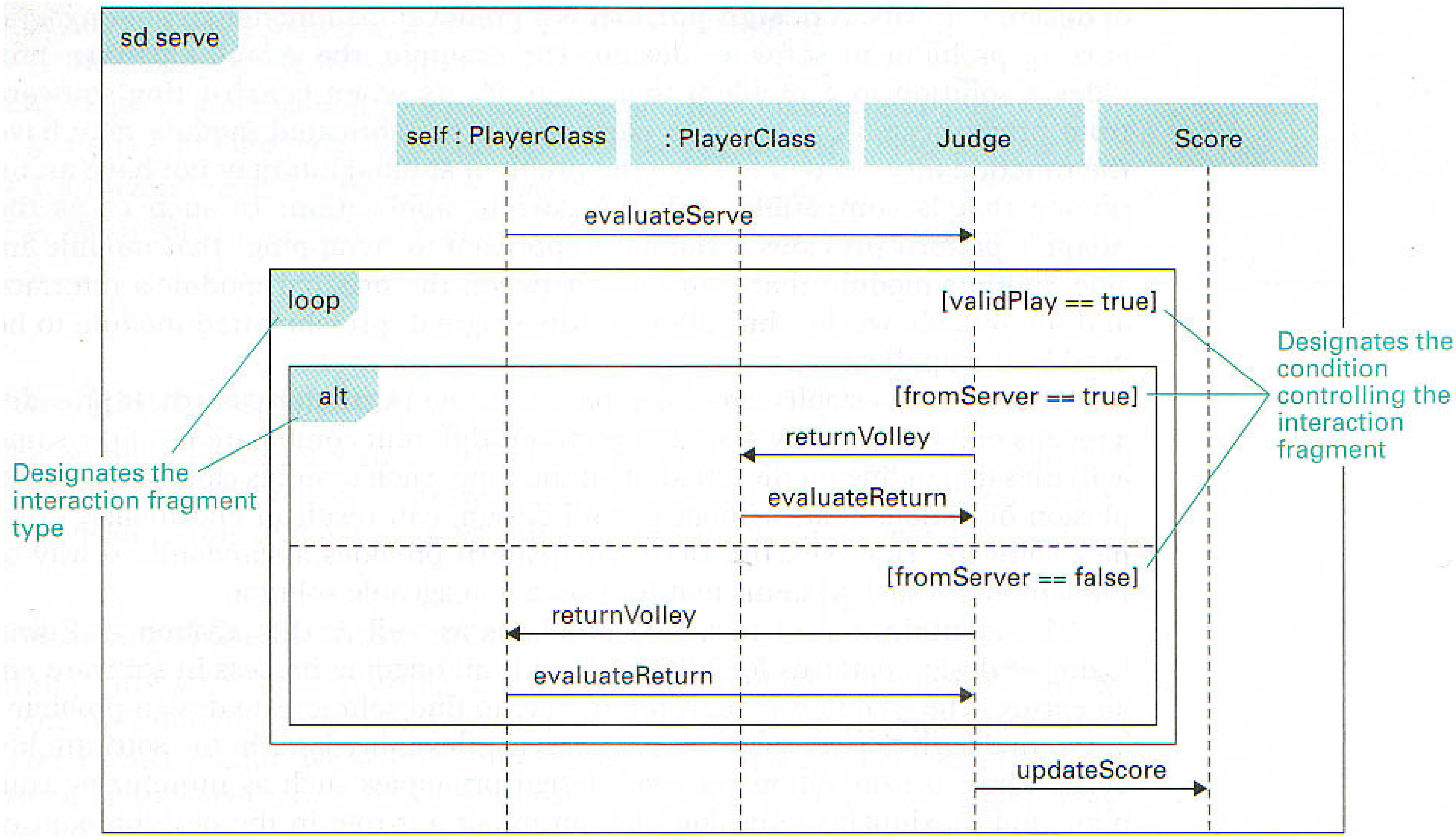
Arrow	Message type
	Synchronous call
	Generic flow of control
	Asynchronous flow of control
	Reply
	Time out call (non-standard)

- Loop and branches



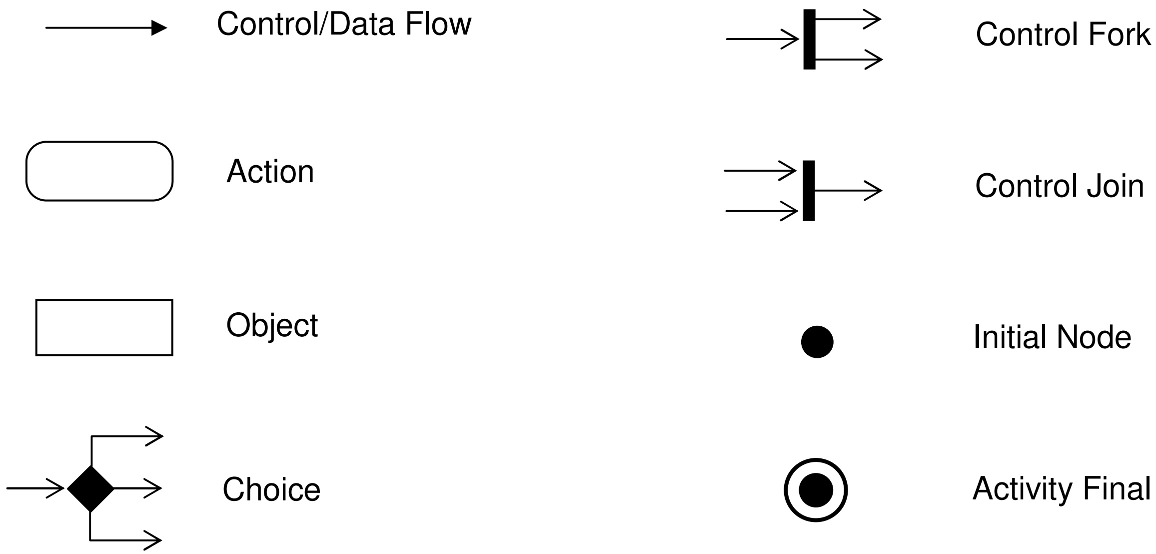


Example of Sequence Diagram



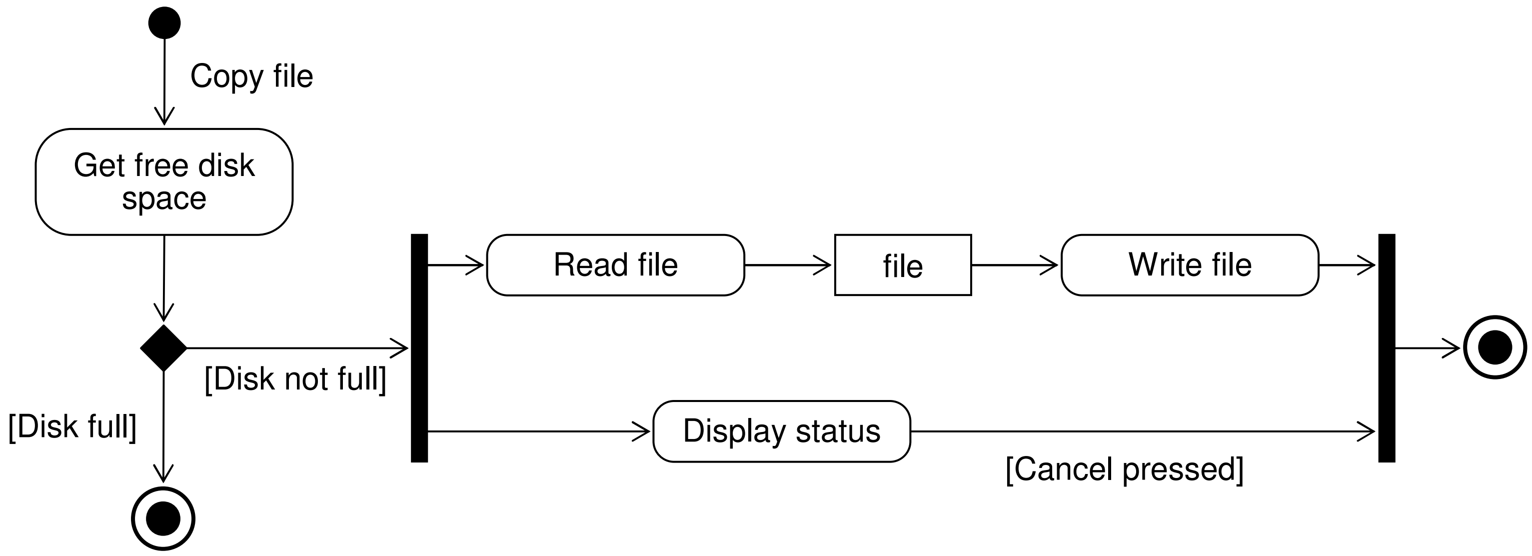
Activity Diagrams (Flow Chart)

- Activity diagrams show flow of control and data flow
  - Notations:



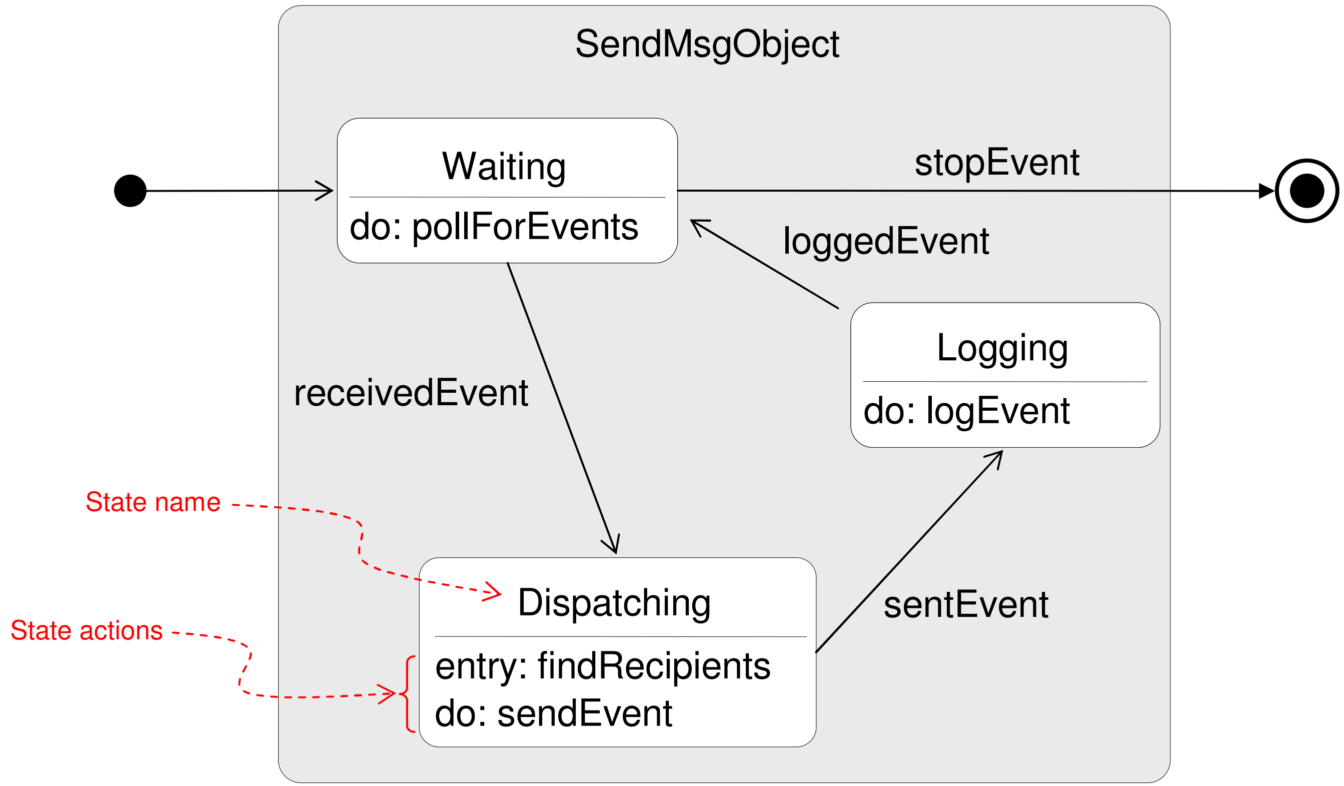
Example of Activity Diagram

- File copy activity can be illustrated as follows:



State Machine Diagram

- State machine diagrams describe the life cycle of an object
- Example: a send-message object



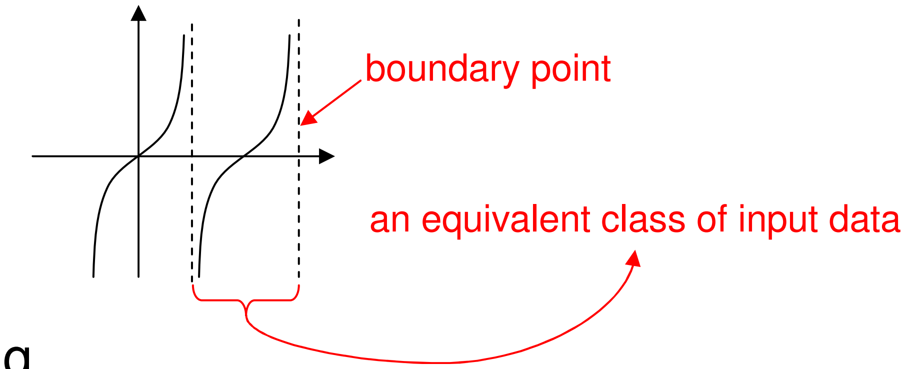
Design Patterns

- Design patterns are "software techniques" for solving recurring problems†
- Examples:
  - Adapter pattern: Used to adapt an existing module's interface to match the interface of a new system
  - Decorator pattern: Used to extend the capability of an existing module so that it can be used in a new system
- Inspired by the work of Christopher Alexander in architecture

† E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, ISBN 0-201-63361-2, 1995.

Software Testing Strategies

- Glass-box testing
  - Pareto principle – typically, only a small amount of software modules in a large system are problematic
  - Basis path testing – test data should enable testing of all possible execution branches in a software system
- Black-box testing
  - Boundary value analysis, e.g. calculation of tan(x)



  - Alpha and Beta testing

Documentation

- User documentation
  - Printed book for all customers
  - On-line help modules
- System documentation
  - Source code is part of the system documentation
    - Consistent coding style and naming conventions
    - Comments
  - Design documents – requirement specifications, algorithm descriptions (in UML, for example), etc.
  - CASE tools can help keep these up to date
- Technical documentation
  - For installing, customizing, updating, etc.

Software Ownership

- Copyright of software
  - Filtration criteria: what is copyrightable?
    - Features covered by International Standards?
    - Characteristics dictated by software purposes?
    - Components in the public domain?
    - Look-and-feel?
  - How do we verify that two software has “substantial similarity”?
- Patents used in software
  - Mathematical formulae are traditionally un-patentable
  - However, some software techniques (algorithms) have become patents → many of them are not defensible in court!
- Trade secrets
  - Non-disclosure agreements are legally enforceable