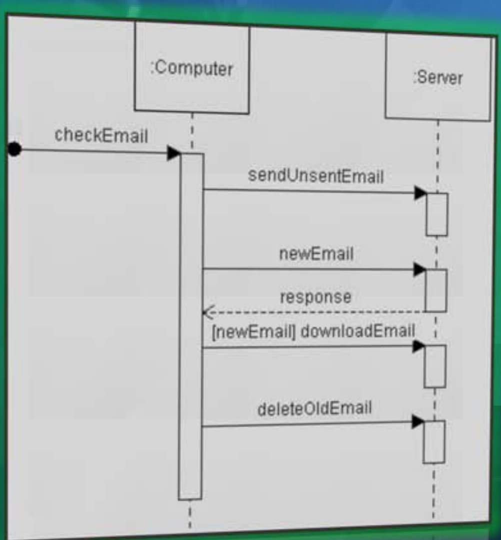
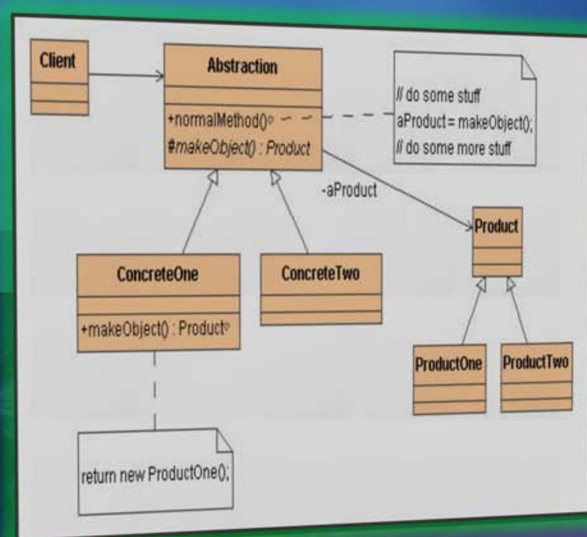
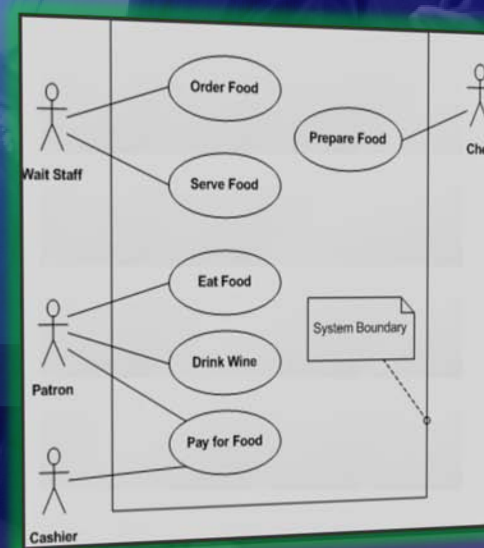
UML

Unified Modeling Language







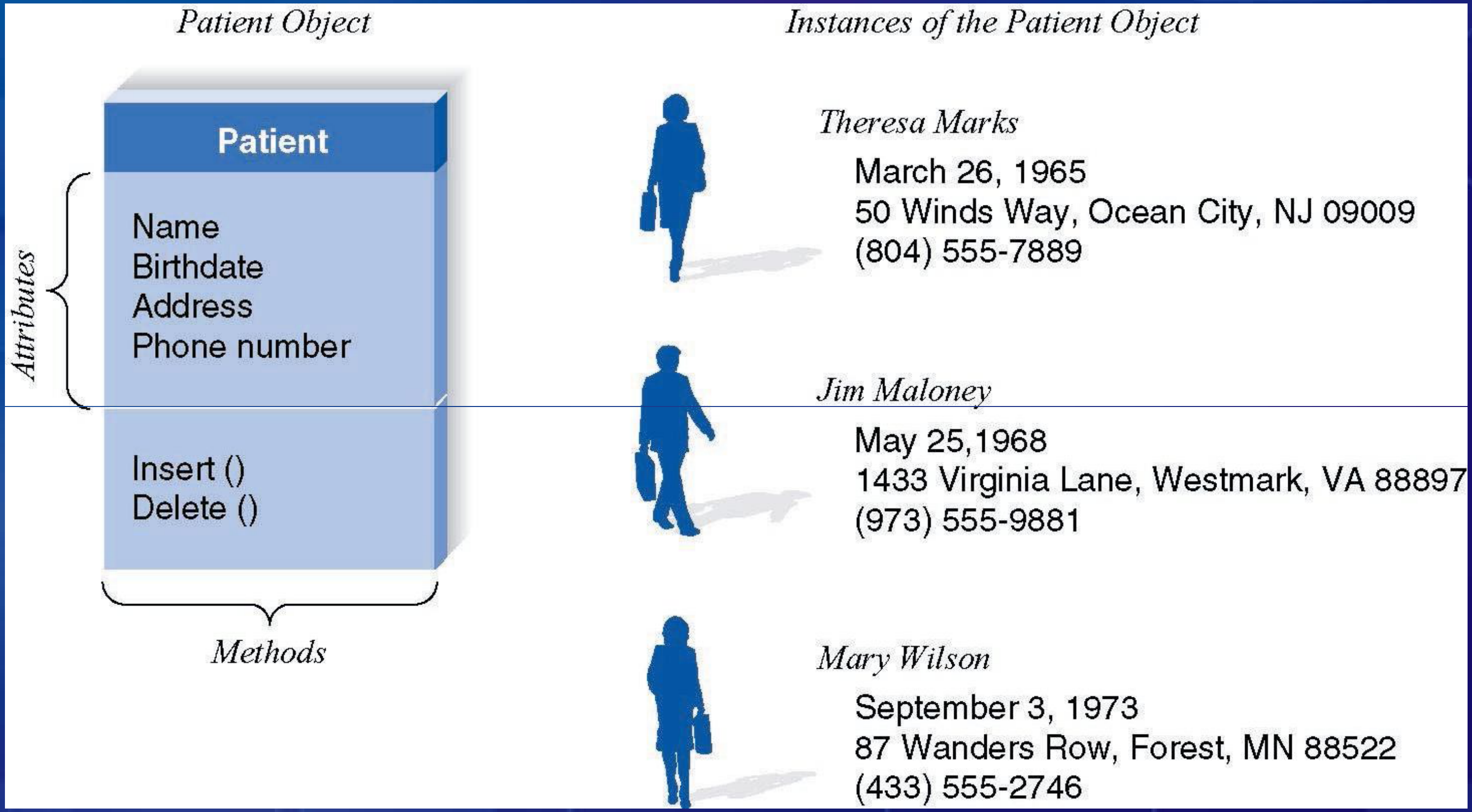
Key Definitions

- Object-oriented techniques view a system as a collection of self-contained objects which include both data and processes.
- The Unified Modeling Language (UML) has become an object modeling standard and adds a variety of techniques to the field of systems analysis and development.

Object Concepts

- **Object**
  An object is a person, place, event, or thing about which we want to capture information.
- **Properties**
  Each object has properties (or attributes).
- **State**
  The state of an object is defined by the value of its properties and relations with other objects at a point in time.
- **Methods**
  Objects have behaviors -- things that they can do – which are described by methods (or operations).
- **Unique**
  Objects do not use primary or foreign keys, instead each instance is assigned a unique identifier (UID) when it is created.

An Object Class and Object Instances

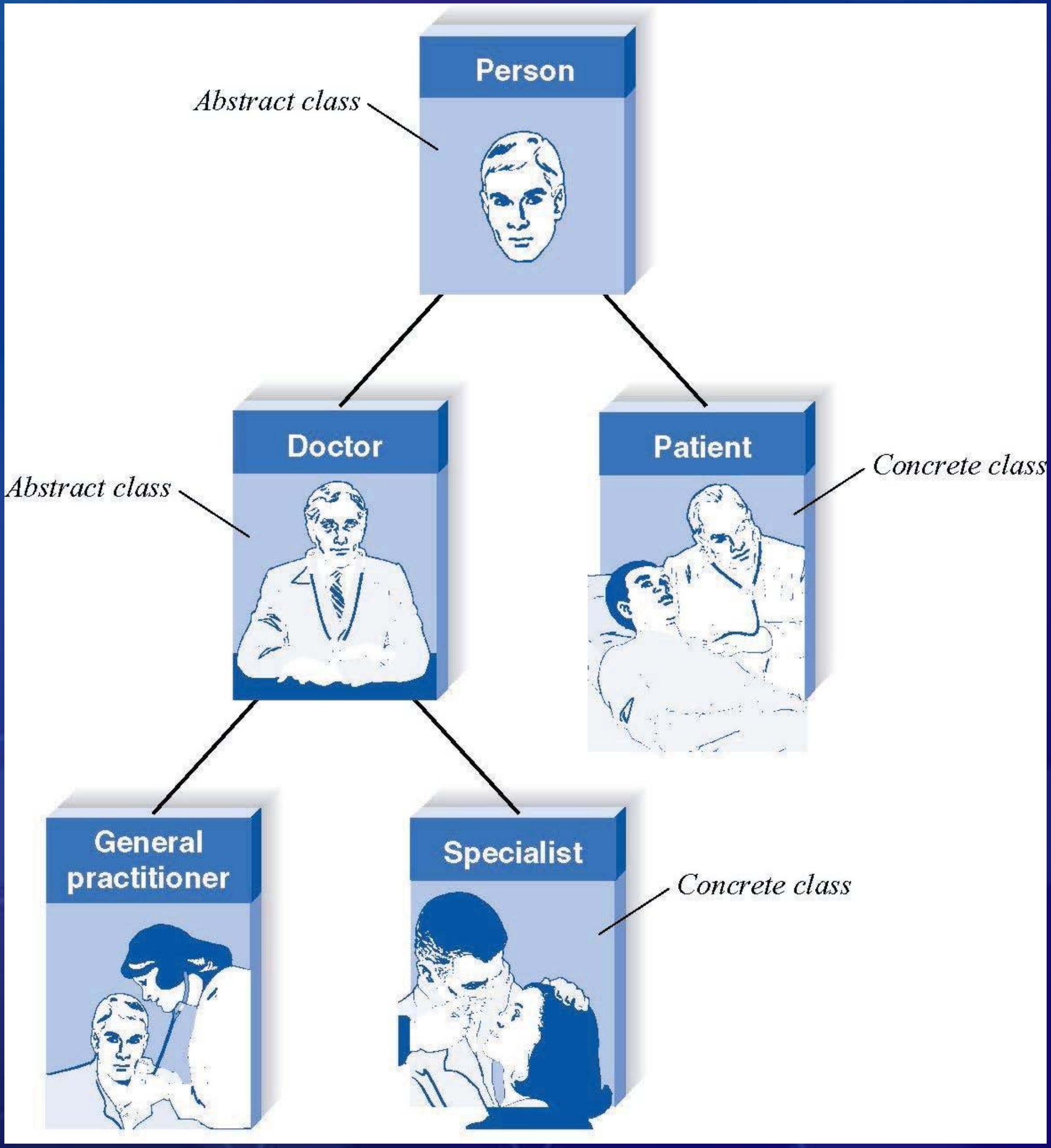


A **class** is a general template we use to define and create specific instances/objects.

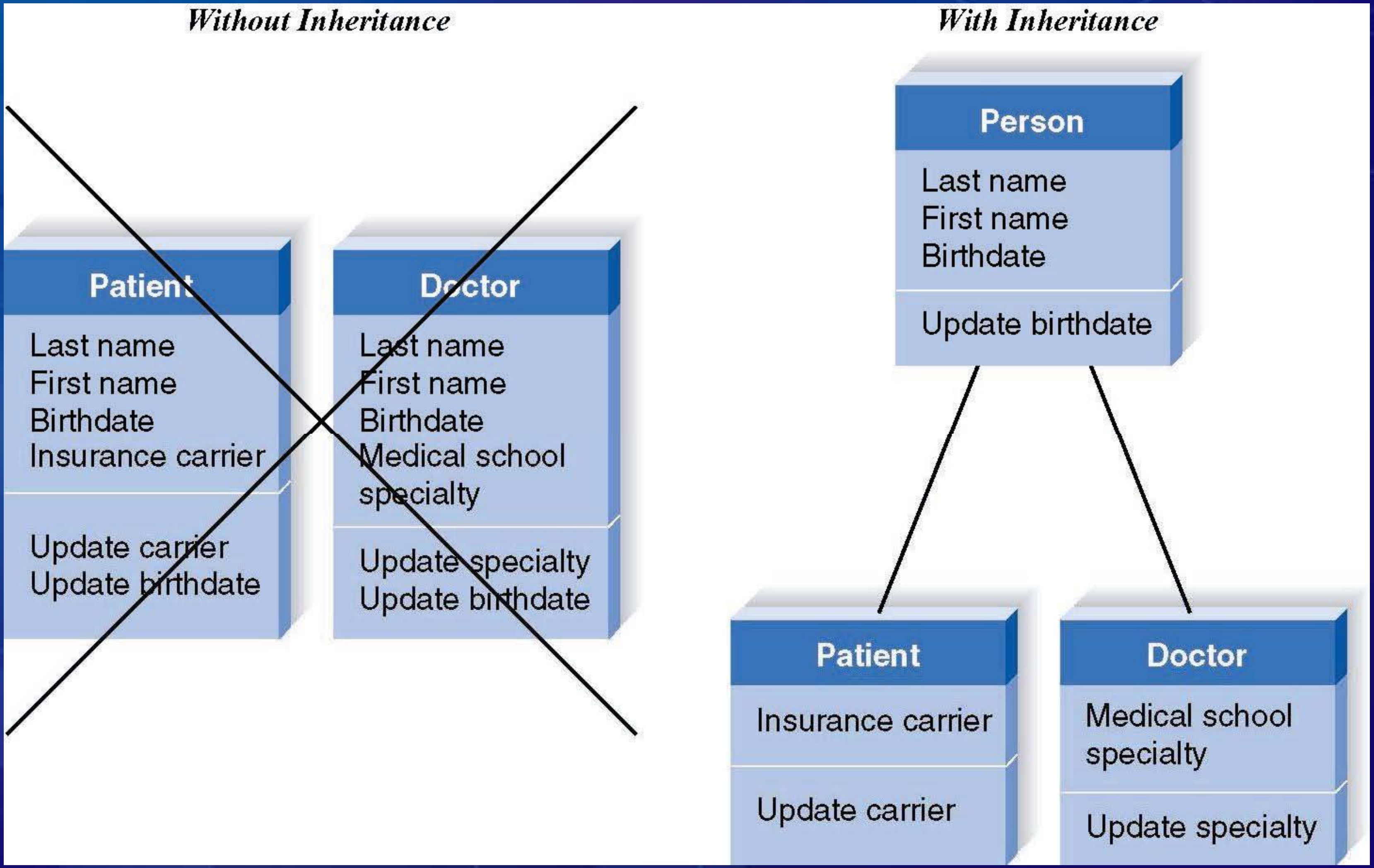
Inheritance

- **Classes are arranged in a hierarchy**
  - Superclasses or general classes are at the top
  - Subclasses or specific classes are at the bottom
    - Subclasses inherit attributes and methods from the superclasses above them
  - Classes with instances are concrete classes
  - Abstract classes only produce templates for more specific classes

Class Hierarchy



Inheritance



Inheritance

designers overuse inheritance (Gang of Four 1995:20)

Encapsulation

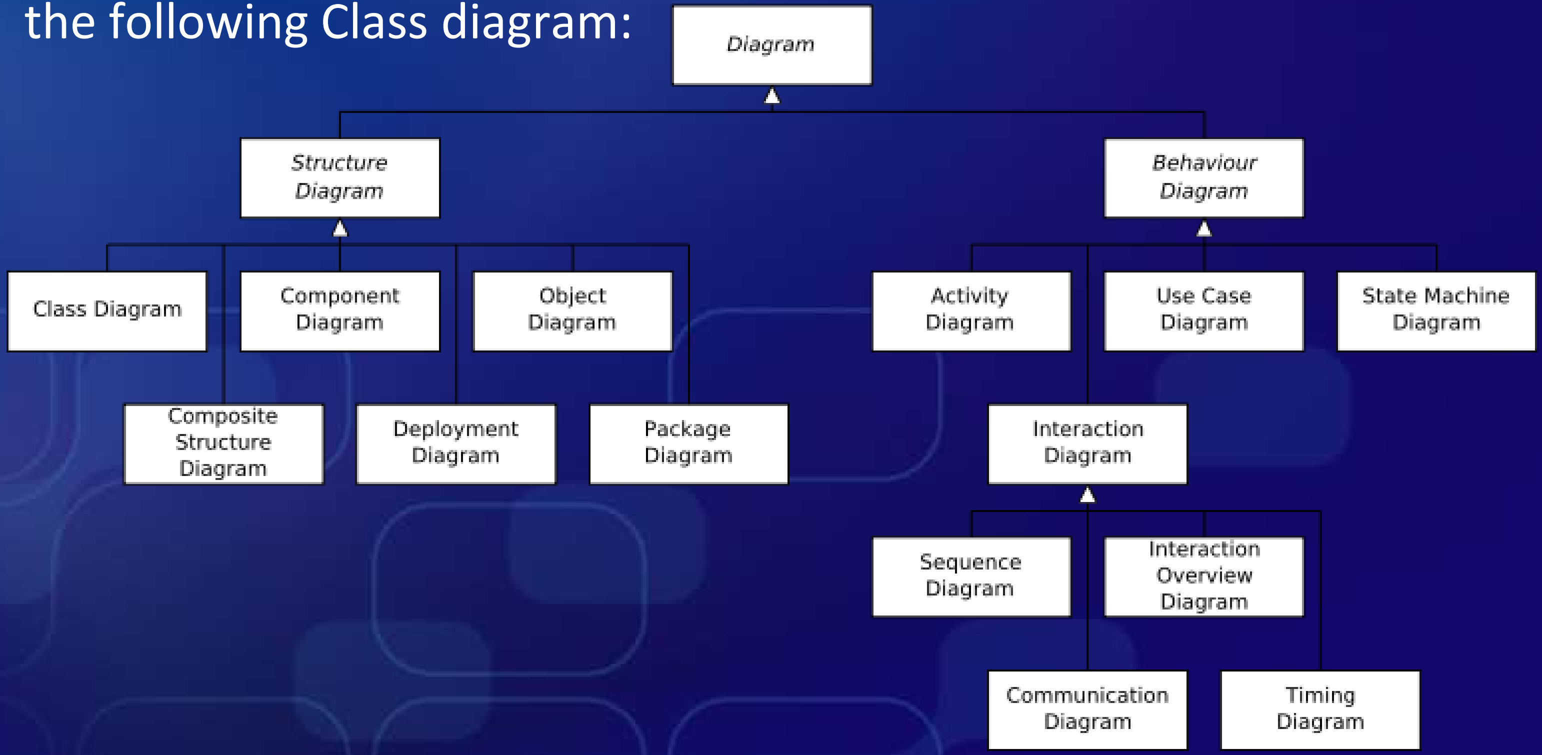
- **The message is sent without considering how it will be implemented**
- **The object can be treated as a "black-box"**
- "Because inheritance exposes a subclass to details of its parent's implementation, it's often said that 'inheritance breaks encapsulation'". (Gang of Four 1995:19)

What is UML

- Unified Modeling Language
- A set of 13 diagram definitions for different phases / parts of the system development
- Diagrams are tightly integrated syntactically and conceptually to represent an integrated whole
- Application of UML can vary among organizations
- The key building block is the Use Case
- Collection of best engineering practices
- Industry standard for an OO software system under development
- Doesn’t mandate a process
- Its not a programming language !! – It’s a way to design the software (modeling language)

What is UML

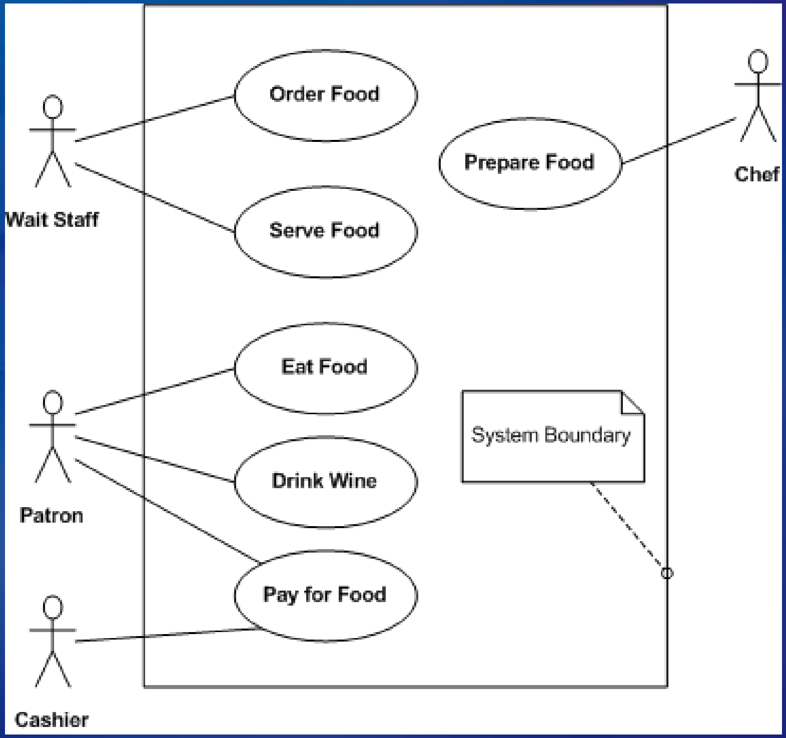
- UML 2.0 has 13 types of diagrams divided into three categories
  - 6 diagram types represent application **<u>structure</u>**
  - 3 represent general types of **<u>behavior</u>**,
  - 4 represent different aspects of **<u>interactions</u>**.
  - These diagrams can be categorized hierarchically as shown in the following Class diagram:

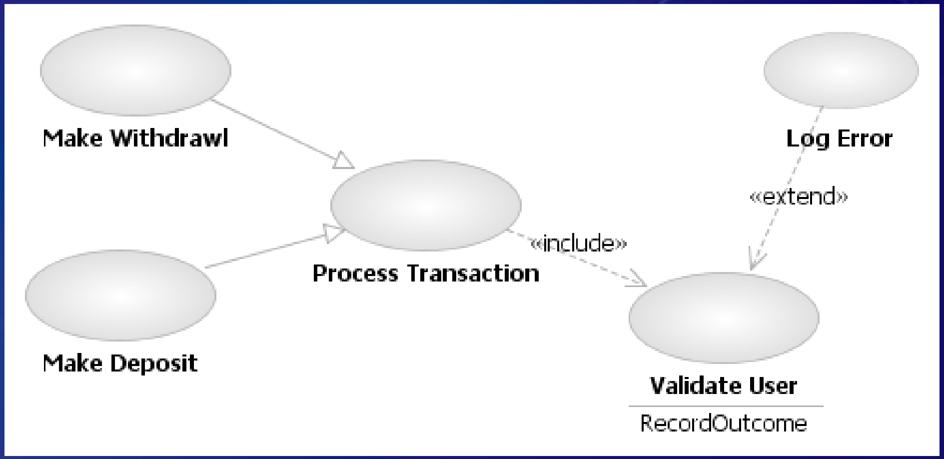


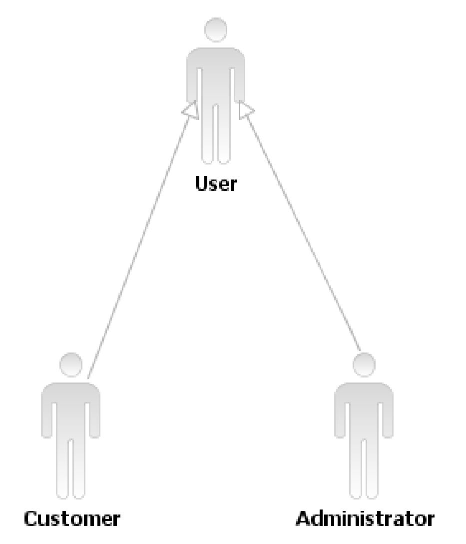
Why using UML?

- Communication between people
- Communication between different roles
- Platform/Technology/Implementation independent
- Visual / Graphical language
- Larger picture of the system (not so detailed as the implementation)
- A good choice for representing and communicating design (and therefore design patterns)

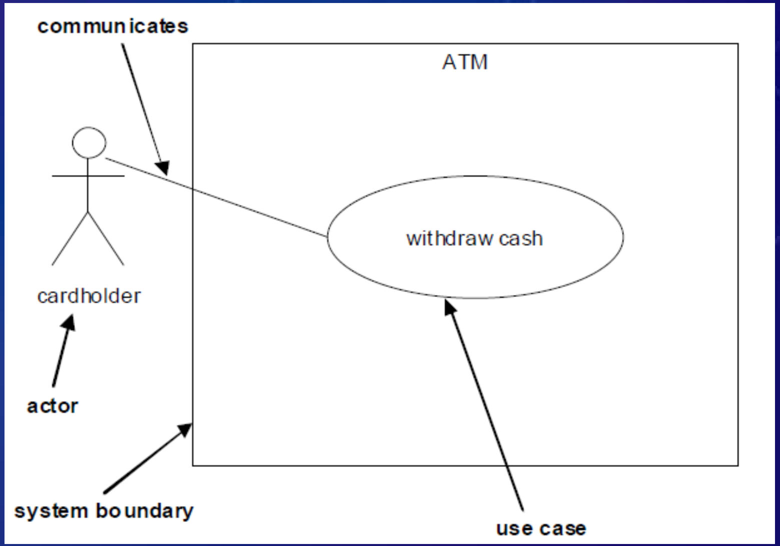
Use Case Diagrams





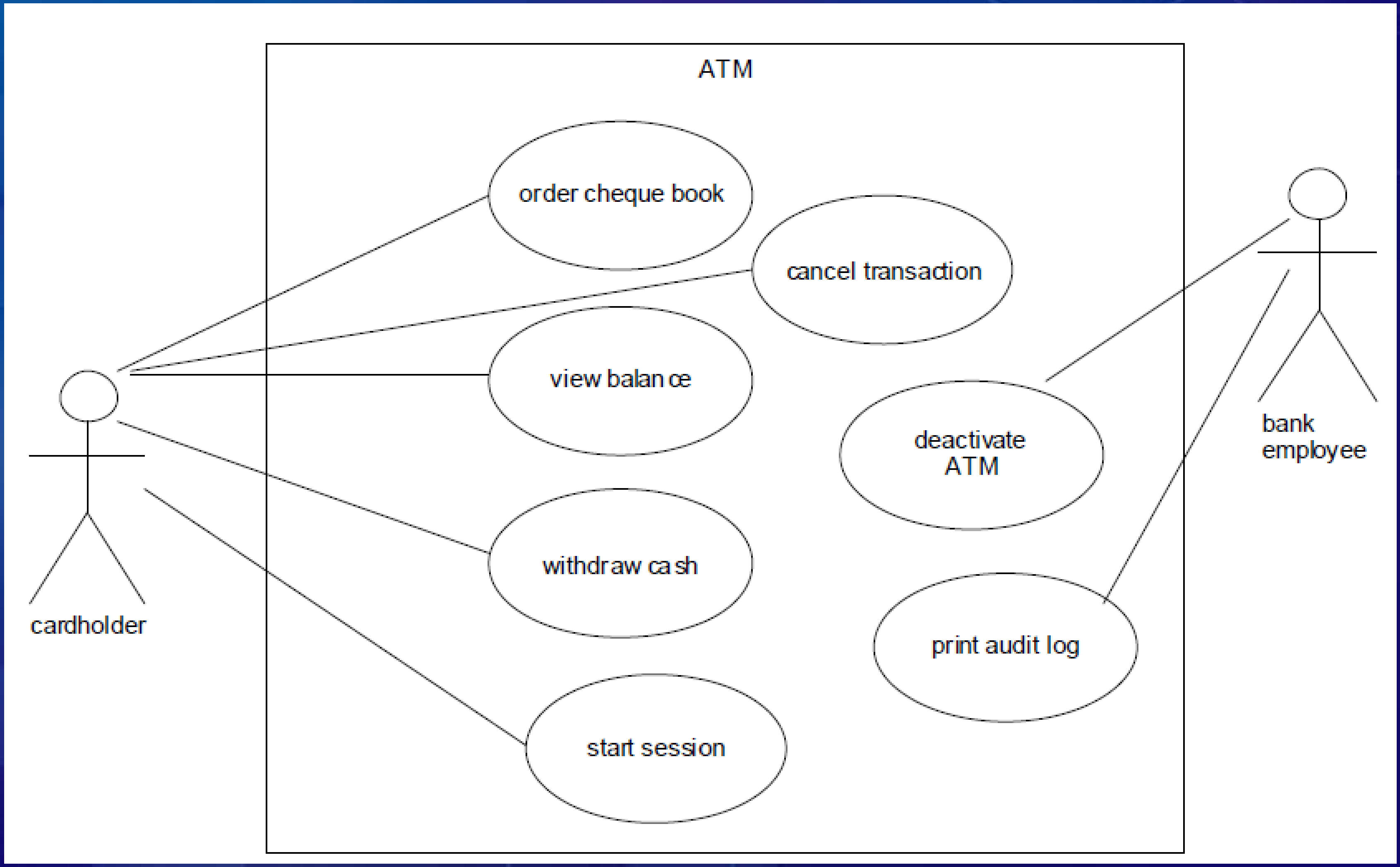


What is a Use Case

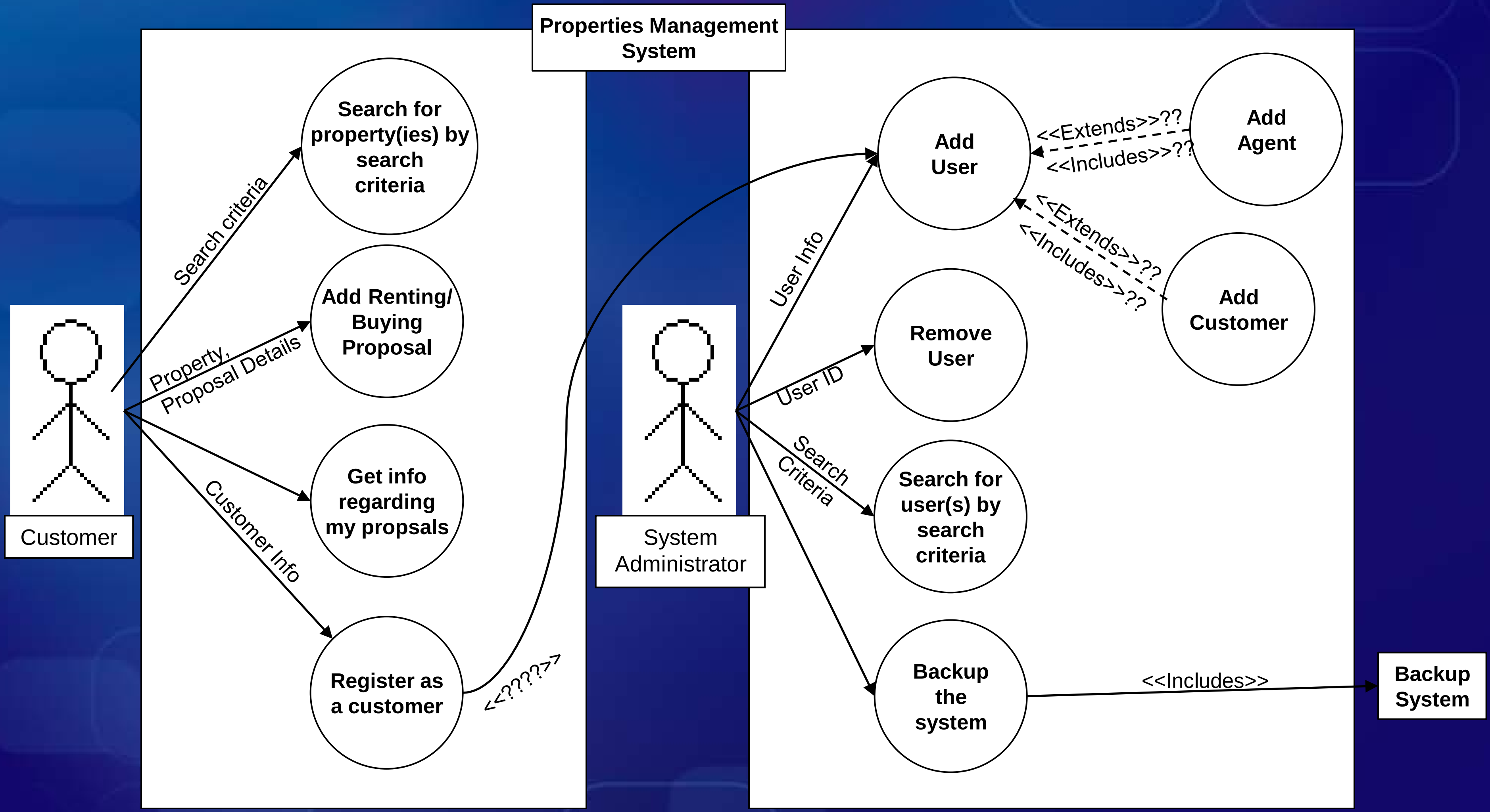


- A reason to use the system
- ATM Example:
  - “Get cash out of the account”
  - “Show balance”
- A Use case is described by:
  - System
  - Actor
    - In relationship with the system (We don’t care that the cardholder is a football player)
    - External to the system it self
    - Doesn’t have to be a person. It can be system that needs services of another system (“ATM” is an actor that uses the “Bank” system)
  - Goal
    - Must be of value to the actor
- Main Two Questions:
  - Who will be using the system?
  - What will they do with it?

Use Case Diagrams



Use Case Diagrams



- Relationship Between Use Cases
  - <<Includes>> - Making coffee always includes boiling water
  - <<Extends>> - Making coffee is sometimes extended by adding sugar

Use Cases Scenarios

Please take your cash...

Sorry. You have insufficient funds. Please specify a smaller Amount.

Sorry, We are unable to process your request at the moment.

Sorry, We are unable to process your request at the moment.

- Same starting point
- Same Need
- Same goal
- <u>Different outcome</u>
- Use cases are defined by key use case scenarios
- Use Case: “Withdraw cash”
  - Scenario 1: Take your cash ☺
  - Scenario 2: Cardholder doesn’t have enough money
  - Scenario 3: ATM has insufficient cash
- The basis of **interaction design**
- Maps to other useful development artifacts
  - UI Storyboards
  - System test scripts
  - User documentation

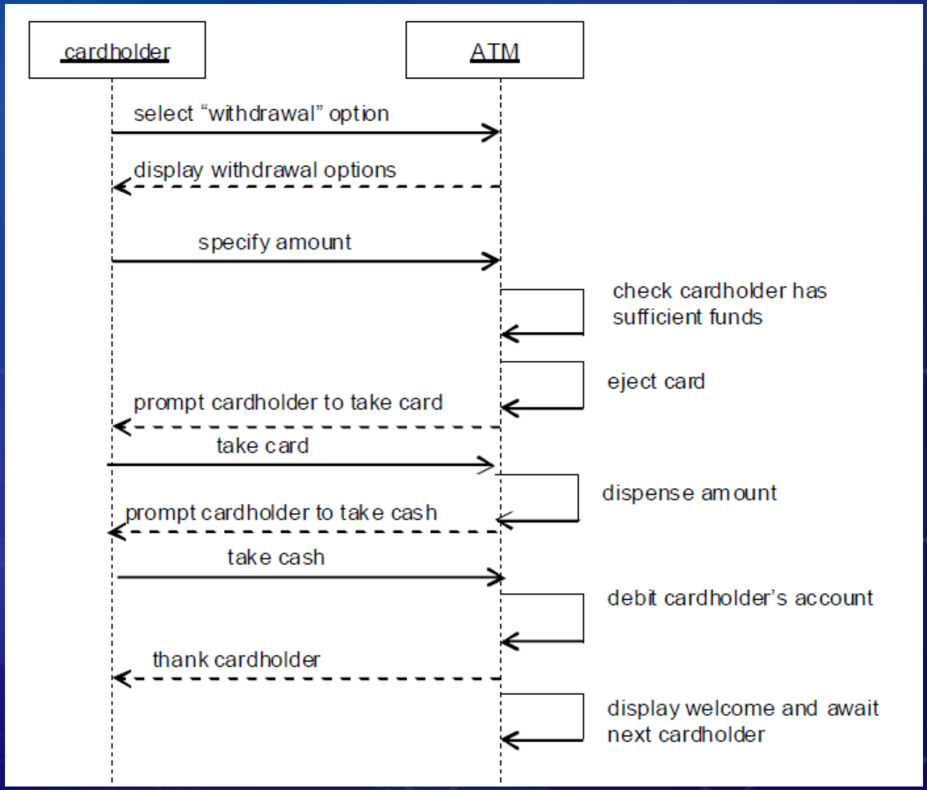
Interaction Design

- Don’t commit to a specific user interface design or implementation technology
- “~~The user presses the ‘enter’ button.~~”
  Instead:
  “The user confirms their choice.”

Cardholder	ATM
•Select “withdrawal” option	
	•Display withdraw options
•Specify amount	
	•Check cardholder has sufficient funds
	•Eject Card
	•Prompt cardholder to take card
•Take cash	
	•Dispense amount
	•Prompt cardholder to take cash
•Take cash	
	•Debit cardholder’s account
	•Thank cardholder
	•display welcome and await next cardholder

Interaction Design

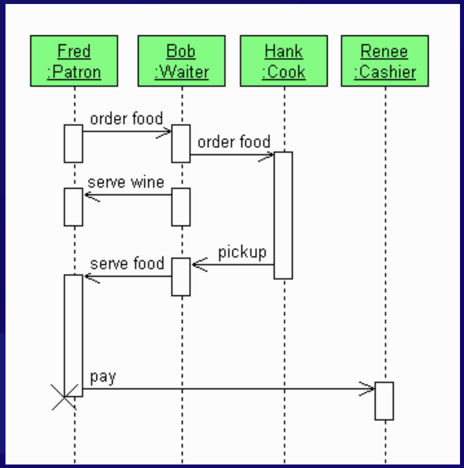
- The basis of high-level OO design, UI design, system test design, user documentation, etc.
- Use case and interaction design ARE NOT the same thing as System Requirements
- The basis for **Sequence Diagrams**:



Sequence Diagrams

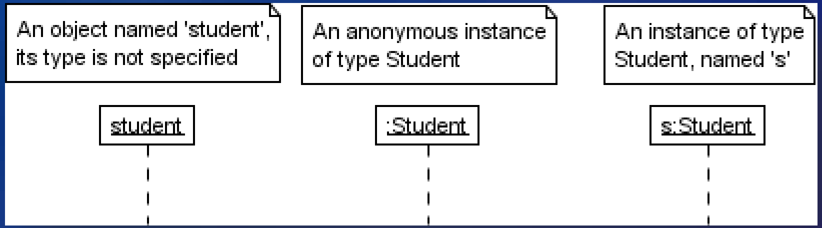
- Illustrates the classes that participate in one use case
- Shows the messages that pass between classes over time for **one** use case
- Drawn for a single scenario in the use case
- Model the behavior of use cases by describing the way group of objects interact to complete a task

- Steps in creating a Sequence Diagram:
  - Identify classes (usually the <u>nouns</u> in the scenario)
  - Add messages (usually the <u>verbs</u>)
  - Place <u>lifeline</u> and <u>focus</u> of control
  - Integrate

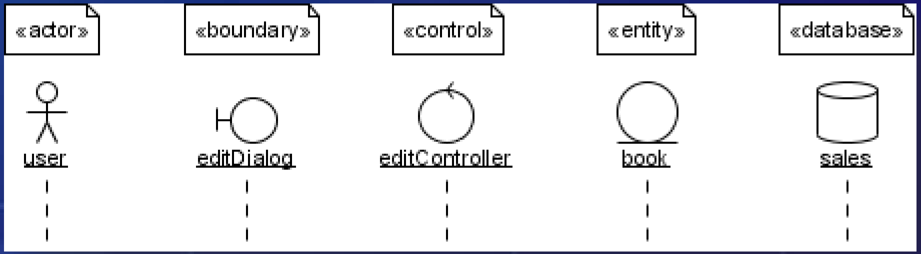


Sequence Diagrams - Syntax

- Targets (objects/classes)
  - Objects
    - Basic notation – a rectangle with an instance name and/or type name, at the top row, with a lifeline under it



    - We can add UML stereotypes to a target and/or icons:

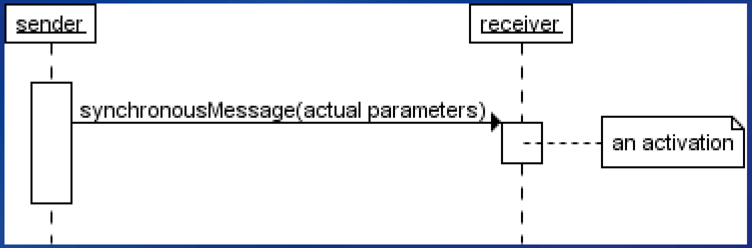


    - Collections

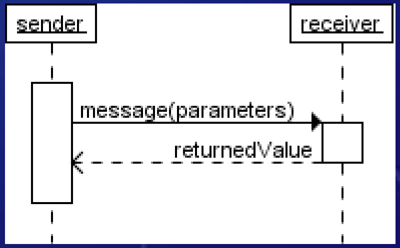
  - Class (for static operations)

Sequence Diagrams - Syntax

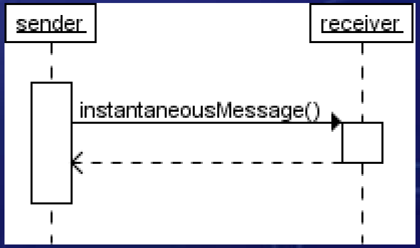
- Messages
  - Synchronous message
    A solid line with a full arrowhead from the sender to the receiver

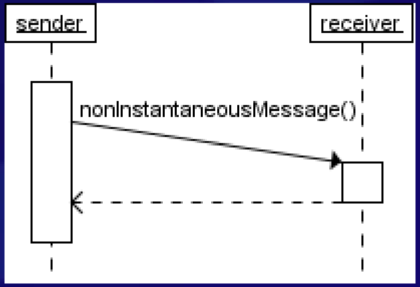


- Return message / value
  A dashed line with an open arrowhead from the receiver back to the caller



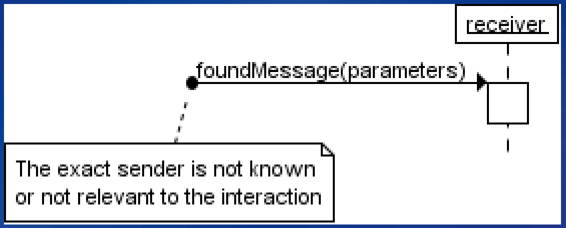
- Instantaneous vs. Non-Instantaneous



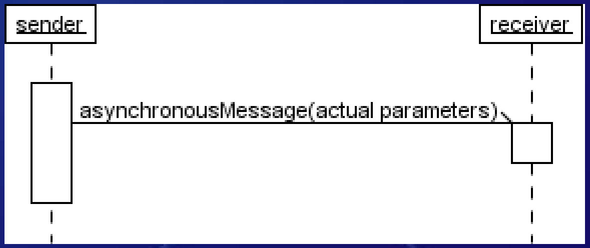


Sequence Diagrams - Syntax

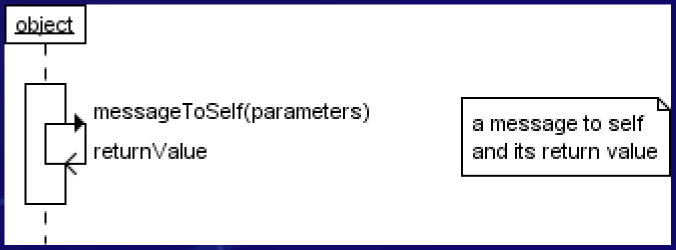
- Messages - Continued
  - 'Found' message
    No caller (either unknown or not important)
    The arrow originates from a filled circle



  - Asynchronous messages
    Half-Open arrowhead

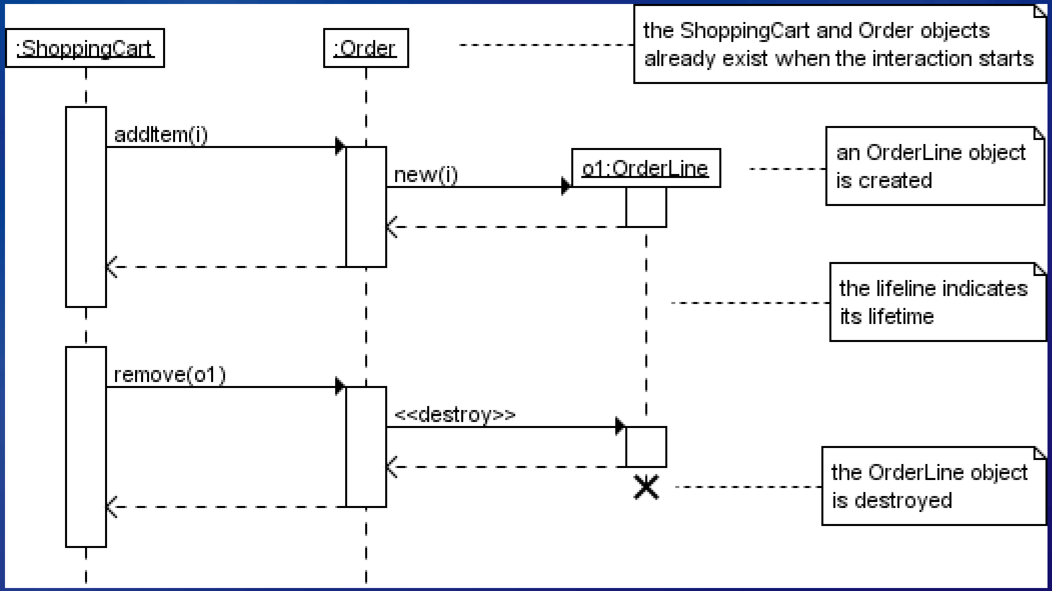


  - Message to self
    keep in mind that the purpose of a sequence diagram is to show the interaction between objects, so think twice about every self message you put on a diagram..



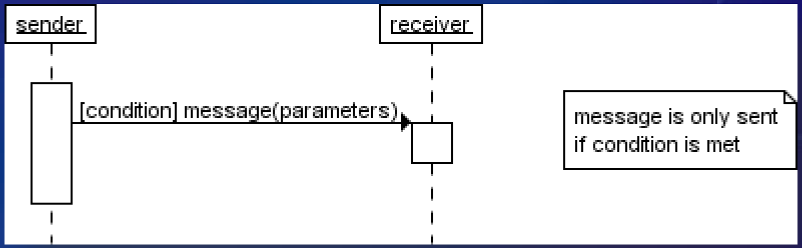
Sequence Diagrams - Syntax

- Messages - Continued
  - Creation and destruction

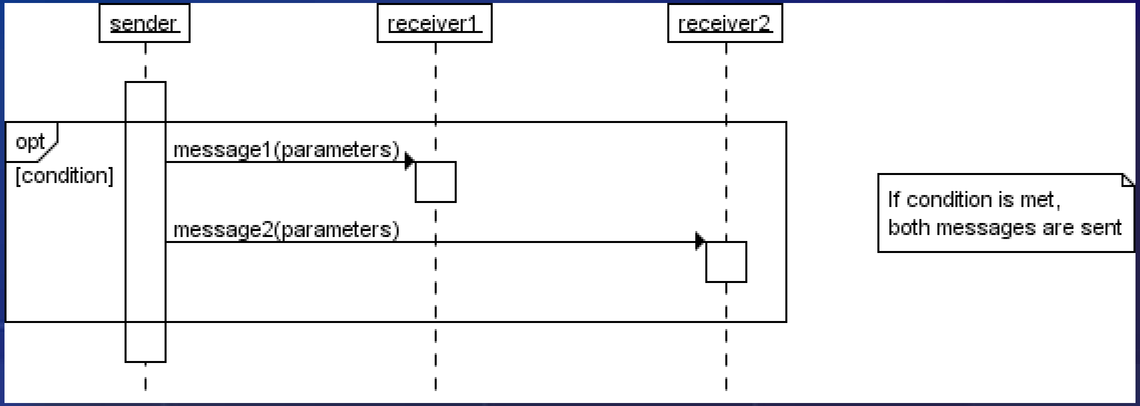


Sequence Diagrams - Syntax

- Conditional Interaction
  - Conditional Message

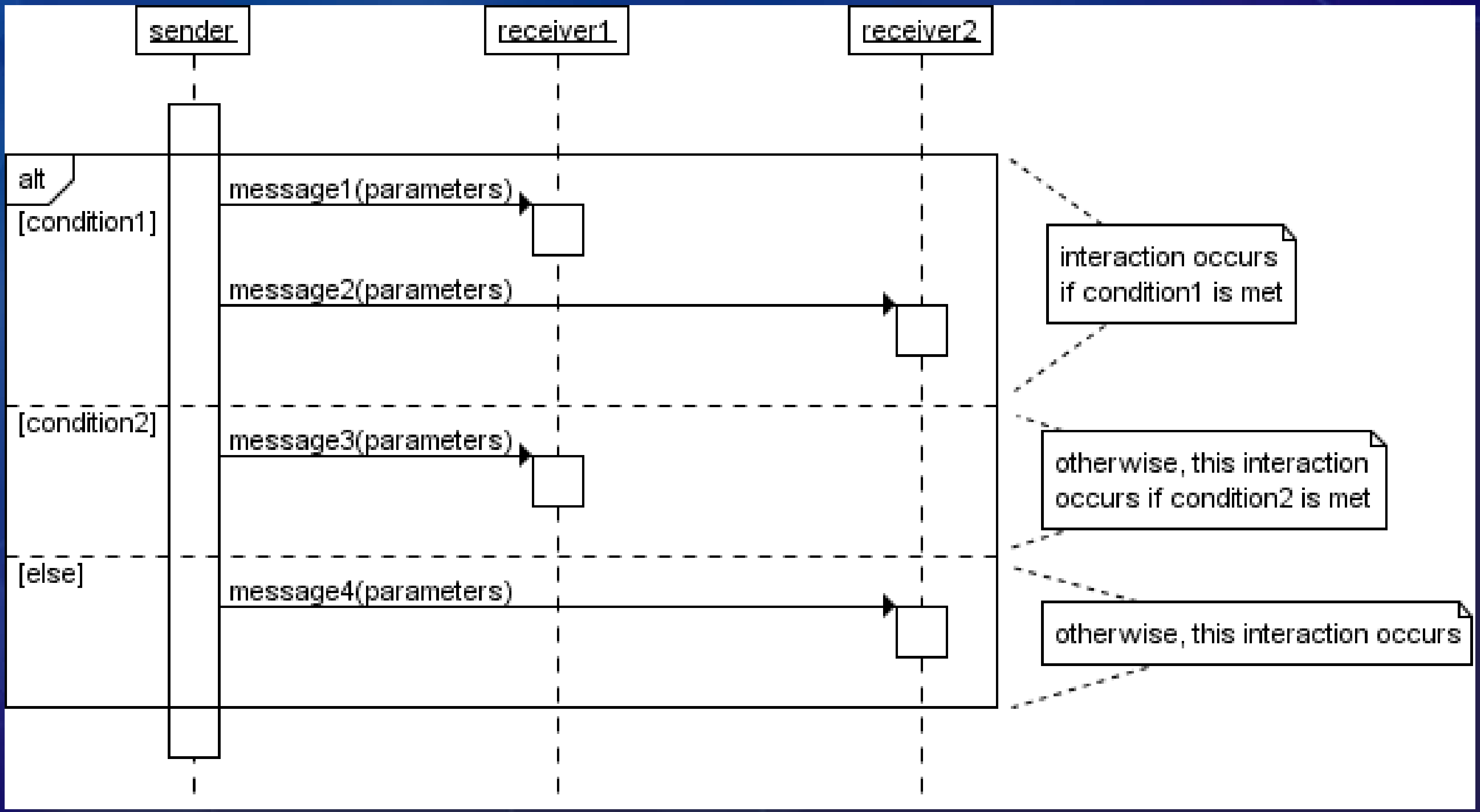


  - Conditional Block



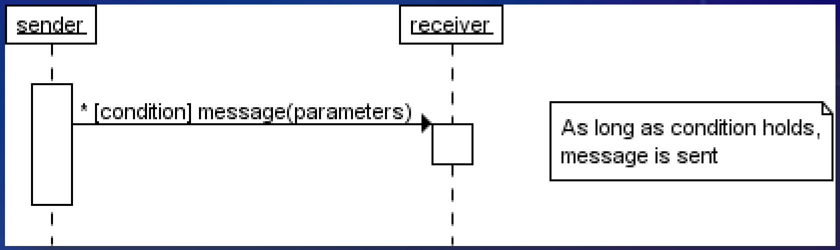
Sequence Diagrams - Syntax

- Conditional Interaction - Continued
  - Alternative Block

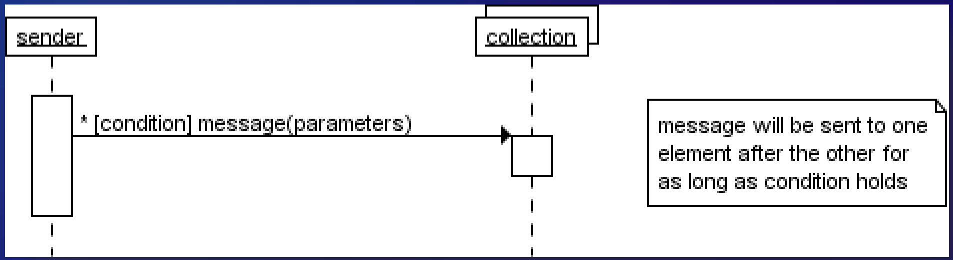


Sequence Diagrams - Syntax

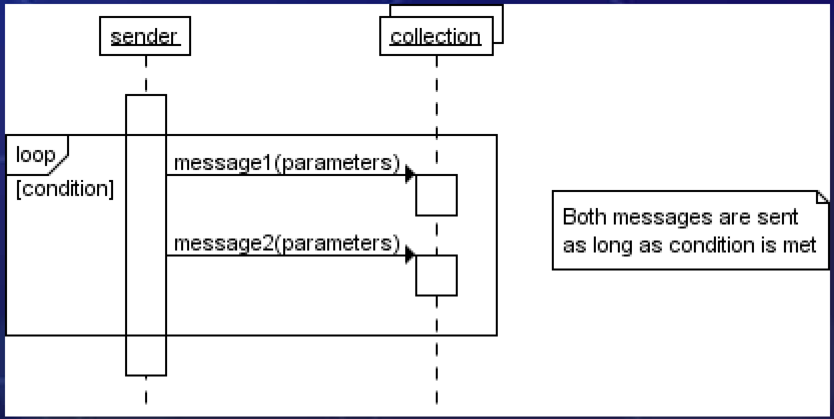
- Repeated Messages
  - Conditional Repeating Message (usually to indicate a polling scenario)



  - Conditional Iterative Message



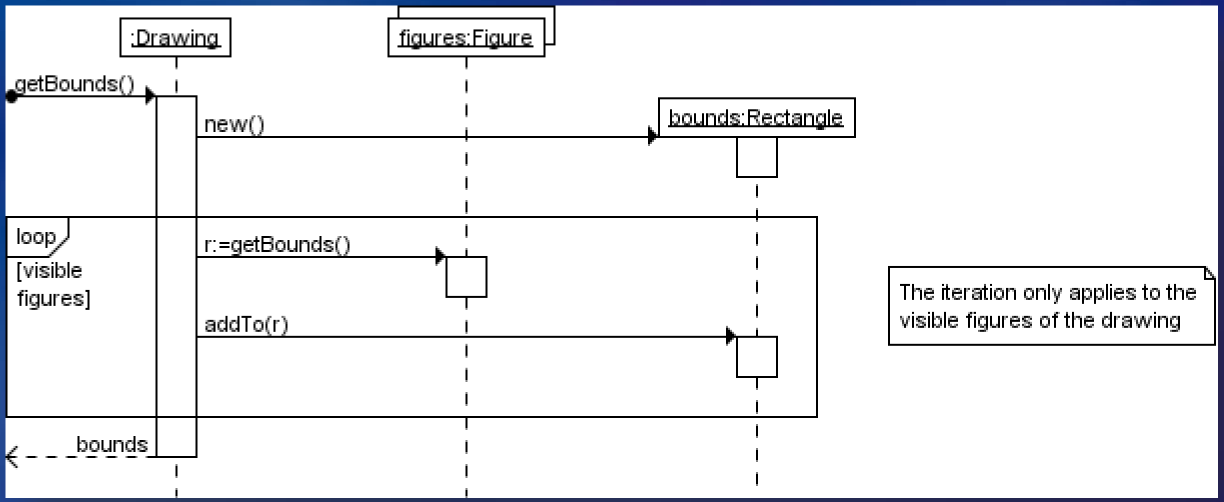
  - Loop block



Sequence Diagrams - Syntax

- Repeated Messages – Example

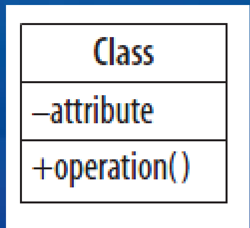
“The bounds of a drawing are based on those of its visible figures”



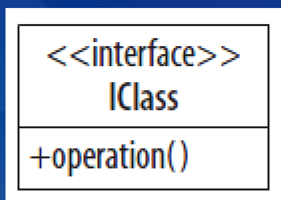
Sequence Diagrams – Keep it agile

- Keep them small and simple
- Their true value is in the creation
- If it's a simple sequence , you can go straight to code
- Use it for complex logic that you want to analyze
- The biggest added-value is realizing the interactions between objects and their lifetime.
- It leads to **class diagrams**

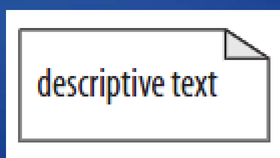
Class Diagrams



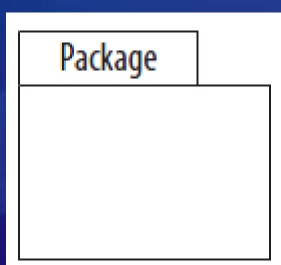
Type
(Class/Struct)
Types and parameters specified when important.
Access indicated by
+ (public), - (private), # (protected).



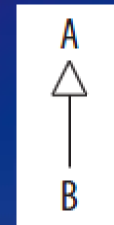
Interface
(and abstract classes)
Name starts with I



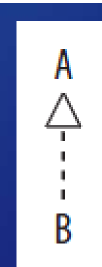
Note
any descriptive text



Package
A library of classes and interfaces
(.NET assmebly)

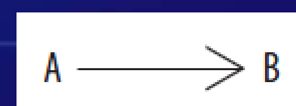


Inheritance
B inherits from A

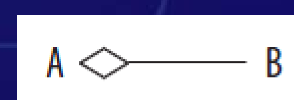


Realization
B implements A

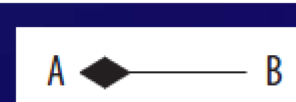
Association
A and B call and access each other's elements.



Association (one way)
A can call and access B's elements, but not vice versa.

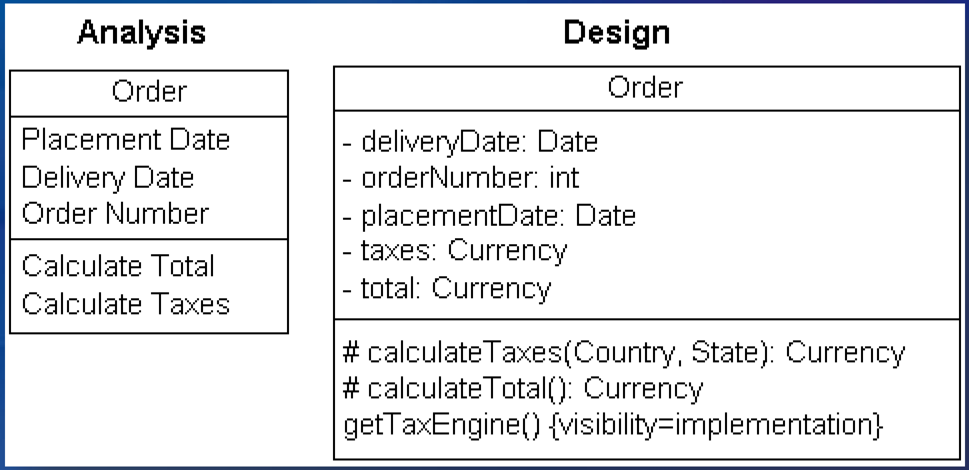


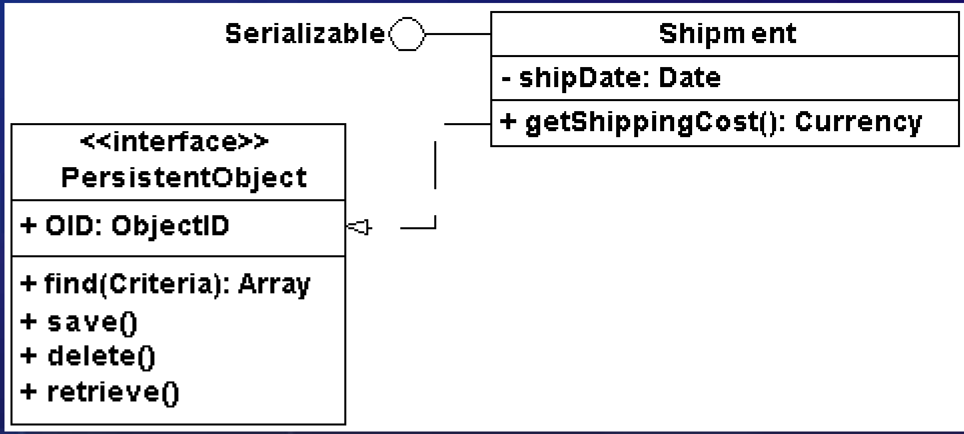
Aggregation
A has a B,
and B can outlive A.



Composition
A has a B,
and B depends on A

Class Diagrams – Analysis vs. Design



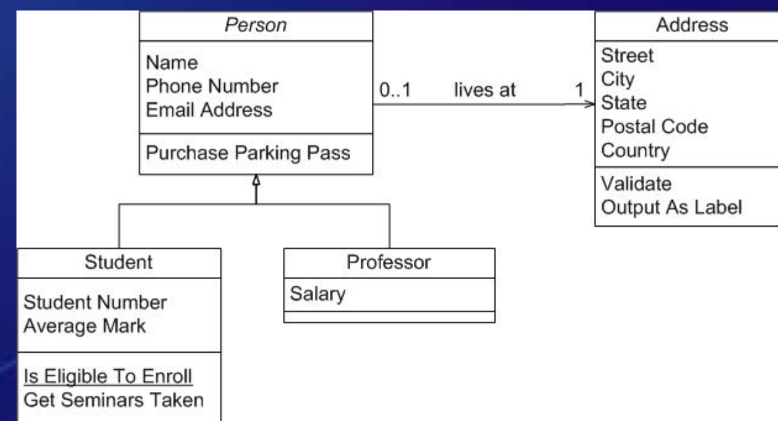


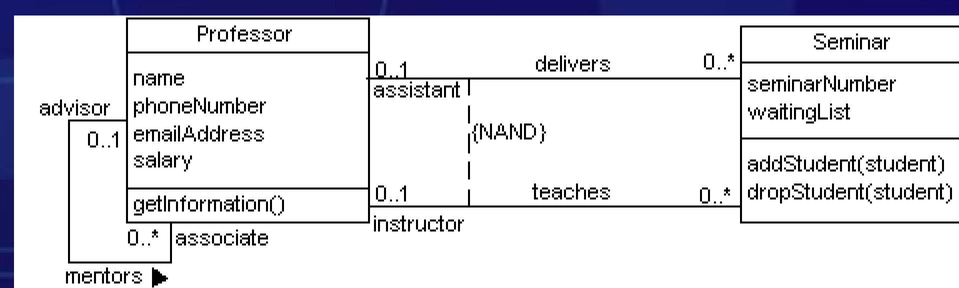
Class Diagrams - Associations

- Notations for associations

- Multiplicity Indicators:

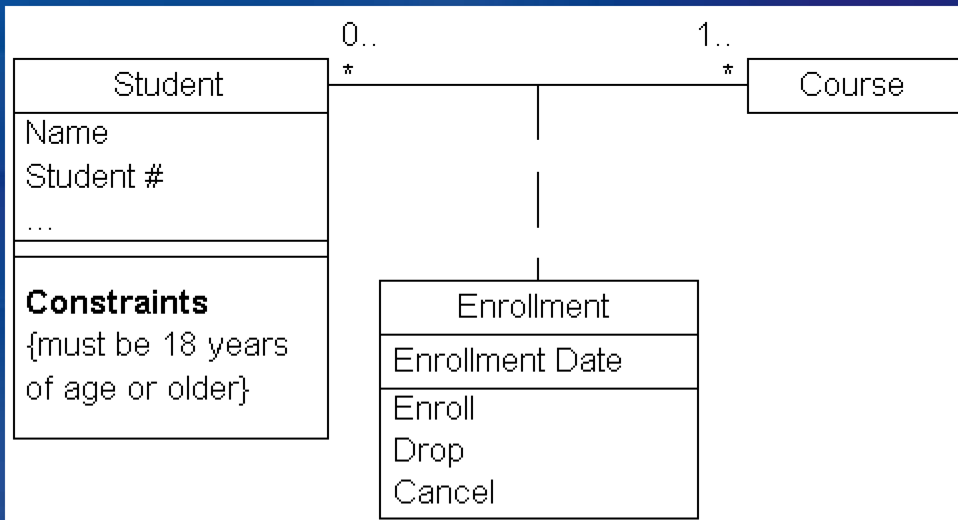
Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only *n* (where *n* > 1)
0..n	Zero to *n* (where *n* > 1)
1..n	One to *n* (where *n* > 1)



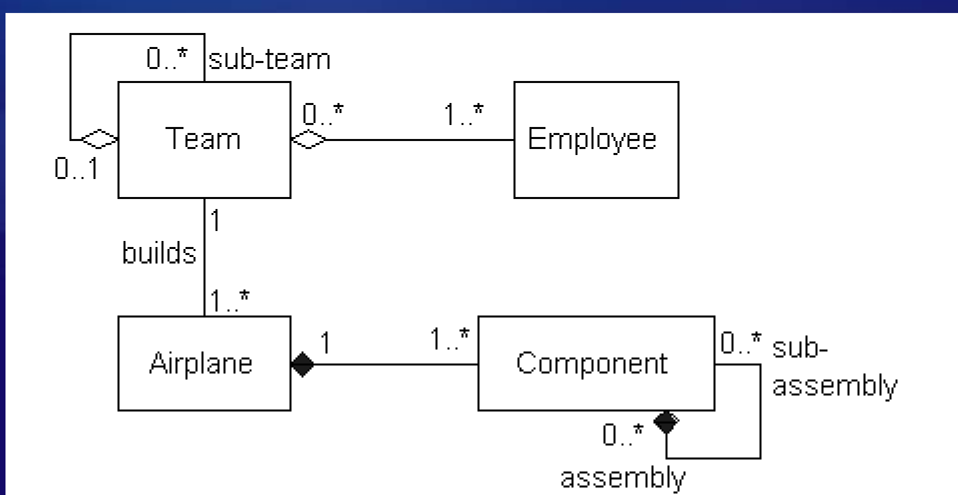


Class Diagrams – Association

- Association class



- Aggregation vs. Composition

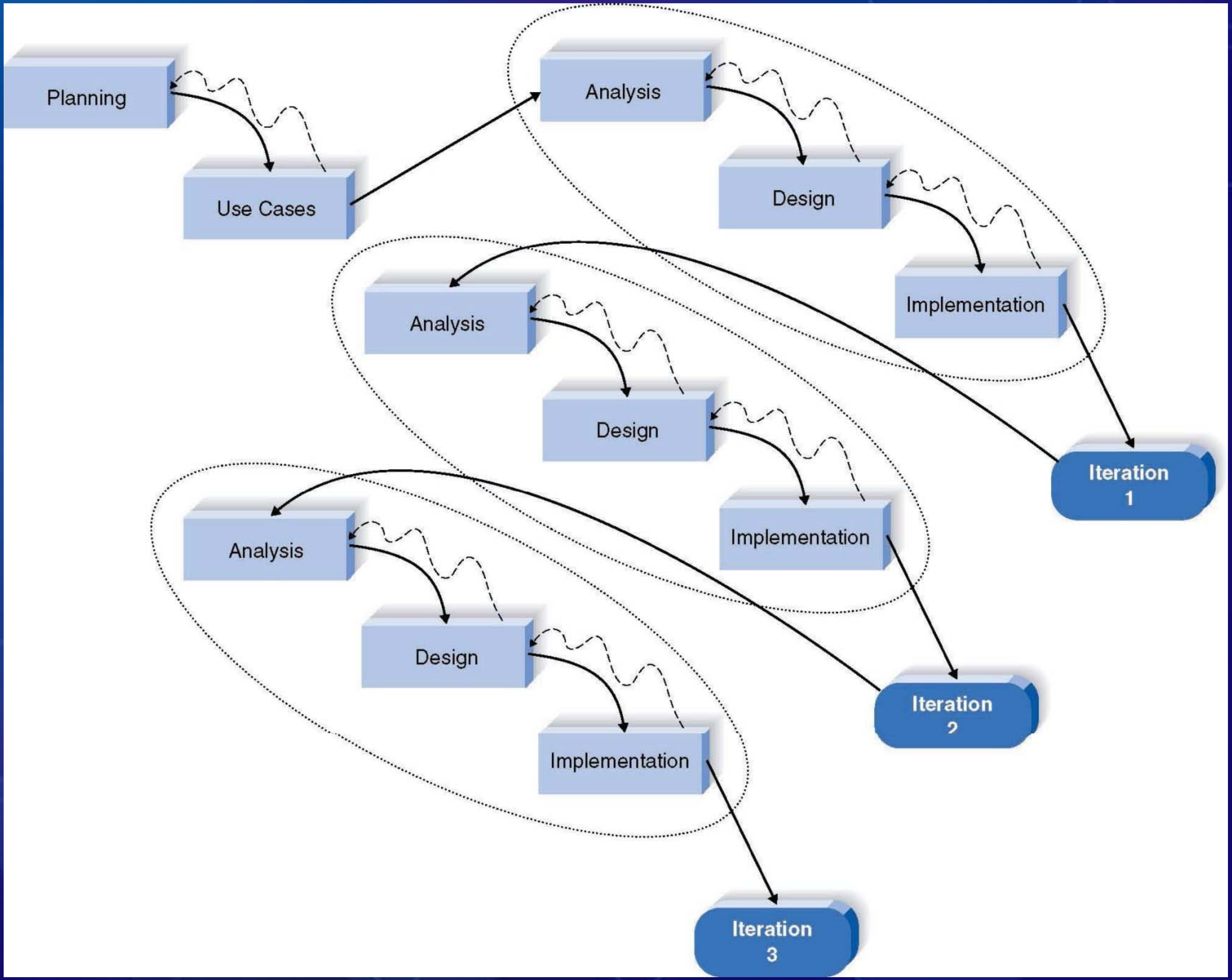


- Both apply the “is part of” relationship
- Depict the Whole to the Left of the Part
- Apply Composition to aggregation of physical items
- Apply Composition When the Parts Share The Persistence Lifecycle With the Whole (usually the hole manage the lifecycle of the parts

UML and Development Lifecycle

- Identify your actors: who will be using the system?
- Identify their goals: what will they be using the system to do?
- Identify key scenarios: in trying to achieve a specific goal, what distinct outcomes or workflows might we need to consider?
- Describe in business terms the interactions between the actor(s) and the system for a specific scenario
- Create a UI prototype that clearly communicates the scenario to technical and non-technical stakeholders
- Do a high-level OO design for the scenario
  - Sequence Diagram, Class Diagrams, Object Diagrams, State
- Implement the design in code
- Get feedback from your users . ideally through structured acceptance testing
- Move on to the next scenario or use case
- **WARNING! Do not, under any circumstances, attempt to design the entire system before writing any code. Break the design down into use cases and scenarios, and work one scenario at a time**

UML in Iterative Development Process



Why not using UML?

- Large and Complex
  - Many diagrams and constructs
  - Redundant and infrequently used
- Weak Visualization
  - Many similar line styles
  - Same line styles can mean different things in different diagram types
- “Only the code is in sync with the code”
- Aesthetically Problematic
- Tries to win them all
- Dysfunctional interchange format