# Object Oriented Design

## Lecture 4

**Ruzanna Chitchyan,** Jon Bird, Pete Bennett

# Overview

- Why would we do OO design?
- OO Design principles
- UML
    - Structure: Class Diagrams
    - Behaviour: Sequence Diagrams
    - Content: Use Cases

# What is OO Design?

Object-Oriented Software: structure software around objects and their interactions.

Objects, as real-world entities, contain **state** and **behaviour**.

Main constructs:

- **Classes**: blueprints for types of objects (e.g., Car)
- **Objects**: individual instances for the given type (e.g., Ford Fusion with registration number AB25CDF )

# Why Design?

- Organise ideas
- Plan work
- Build understanding of the system structure and behavior
- Communicate with development team
- Help (future) maintenance team to understand

# OO Design Principles

# OO Design Principles: **Encapsulation**

- Encloses the data and behaviour of the objects within that object
- Prevents unauthorised access
    - Controlled access:  object can be changed only by permitted methods
    - Protects data integrity

```
//Partial example, not full class

public class Animal {

   private String name;

   public String getName() {
        return name;
    }
}
```

# OO Design Principles: **Abstraction**

- Focus on core concerns
- Leave out unnecessary detail
- Expose only the essential information
- Hide complexity of the class/object (i.e., how they are made)
    - Loose coupling:  objects interact via abstract interfaces

```
//Partial example, not full class

public abstract class Animal {
   private String name;
   public String getName() {
       return name;
 public abstract void makeSound();
}
+++++++++++++++

class Cat extends Animal {
  public void makeSound() {
        System.out.println ("Meow!");
    }
}

class Dog extends Animal {
  public void makeSound() {
        System.out.println (”Woof!");
    }
}
```

# OO Design Principles: **Inheritance**

- Inherit the properties and behaviour
- Reuse the code
  - Reduce re-writing
  - Reduce errors and inconsistency in code

```
//Partial example, not full class

public abstract class Animal {
   private String name;
   public String getName() {
         return name;
 public abstract void makeSound();
}
+++++++++++++++

class Cat extends Animal {
  public void makeSound() {
         System.out.println ("Meow!");
    }
}
```

# OO Design Principles: **Polymorphism**

- Allows for substitution of sub-class for super-class
- Behaviour belongs to the sub-classed object
  - Dynamic method dispatch
  - Extensibility
  - Flexible and modular systems

```
//Partial example, not full class

public class Main {

    public static void main(String[] args) {

    Animal animal1 = new Dog();
    Animal animal2 = new Cat();

    animal1.makeSound(); // Output: Woof!
    animal2.makeSound(); // Output: Meow!
    }
}
```

# OO Design Principles: **Composition**

- Build an object from other objects
  - Reuse
  - Build incrementally

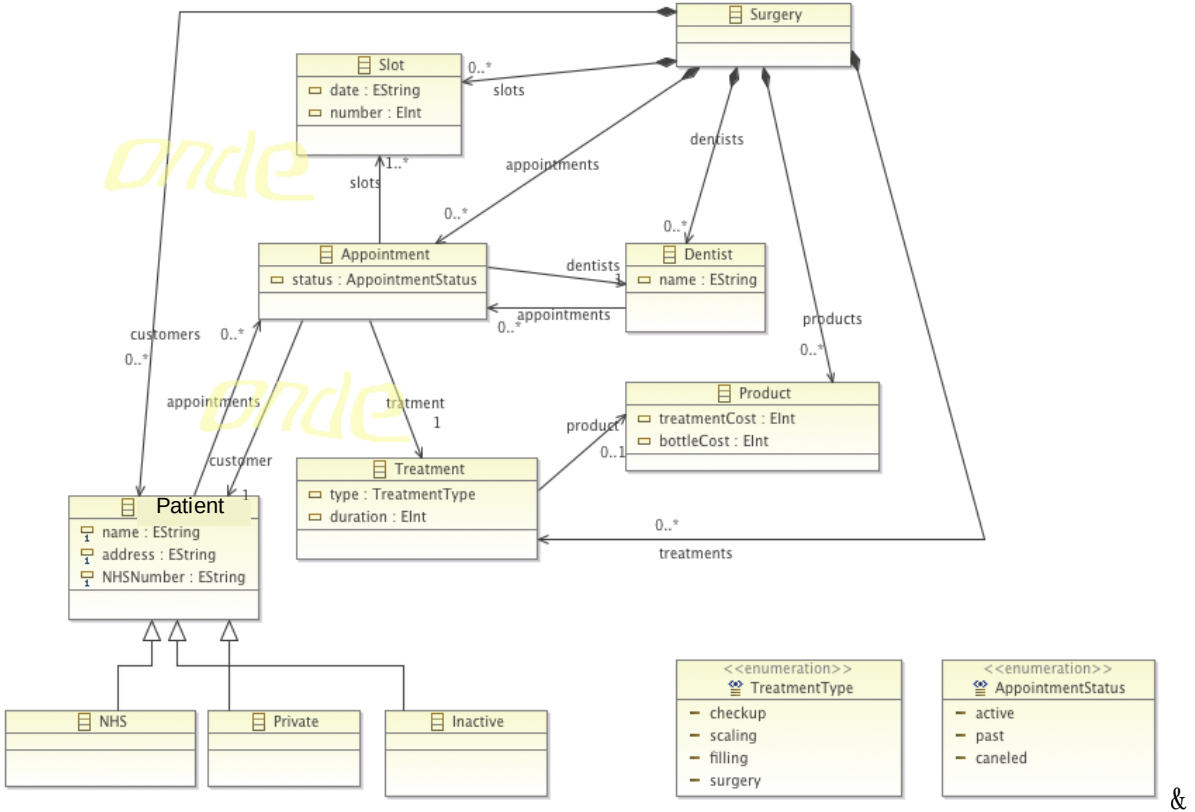
```
//Partial example, not full class

class Legs {
      private int nuOfLegs;
      public Legs (int nuOfLegs) {this.nuOfLegs =nuOfLegs;
      }
      public void walk() {
      System.out.println ("Walking on" + nuOfLegs +" legs");
      }
 }
++++
public class Animal { …
     private Legs legs;
     …
     public void move() { System.out.print(name + " is ");
     legs.walk(); }
}
```

# UML: Class Diagrams

# What Is a Class Diagram?

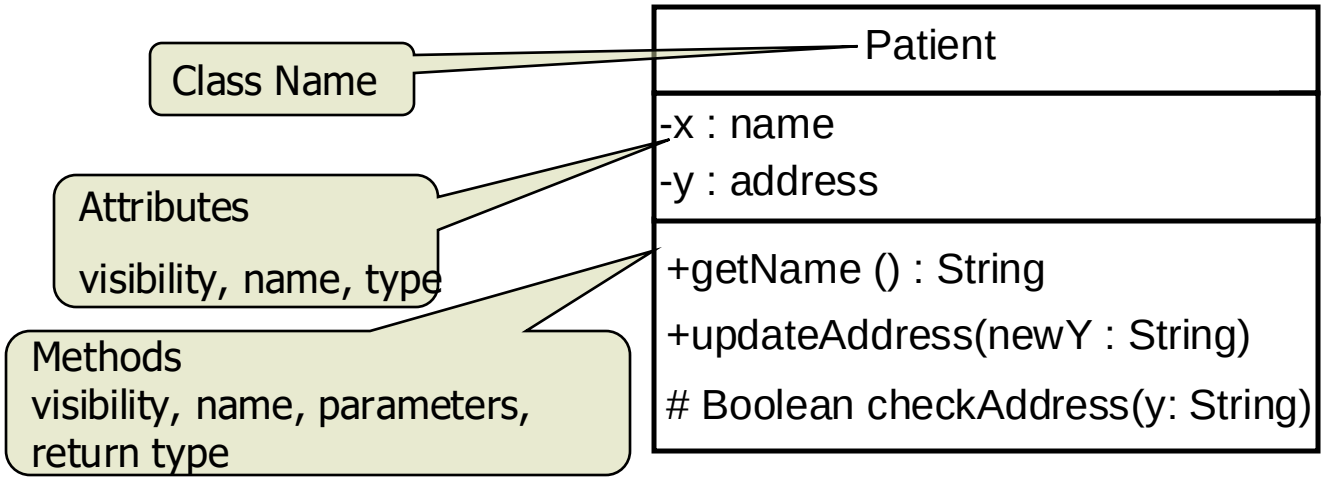
- Static view of a system




&

# Class Diagrams

Class can be understood as a template for creating objects with own functionality




Visibility:

+ public  # protected  - private

# Notation for Attributes

[visibility]  name [: type] [multiplicity] [=value] [{property}]

- visibility
  - other package classes
  - - private : available only within the class
  - + public: available for the world
  - # protected: available for subclasses and other package classes
  - ~ package: available only within the package
- [multiplicity], by default 1
- properties: readOnly, union, subsets<property-name>, redefined<property-name>, ordered, bag, seq, composite
- static attributes appear underlined

# Notation for Operations

[visibility] name ([parameter-list]) : [{property}]

- visibility
- method name
- formal parameter list, separated by commas
  - direction name : type [multiplicity] = value [{property}]
  - static operations are underlined
- Examples:
  - display()
  - - hide()
  - + toString() : String
  - + createWindow (location: Coordinates, container: Container): Window {readOnly}

# How do we find Classes: Grammatical Parse

Classes

- Identify nouns from existing text
- Narrow down to remove
    - Duplicates and variations (e.g., synonyms)
    - Irrelevant
    - Out of scope

# Grammatical Parse: Dental Surgery Example

You are responsible for development of a software system for keeping track of the appointments and services of a dental surgery. This business employs several dentists, provides treatments to NHS and non NHS patients, and allows for patients to buy products (e.g., toothbrush, paste, etc.) when they pay for the received services (such as periodontal therapy with plague removal and scaling, and dental surgery).

# Grammatical Parse: Dental Surgery Example

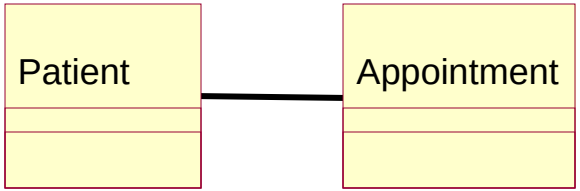
You are responsible for development of a software system for keeping track of the appointments and services of a dental surgery. This business employs several dentists, provides treatments to NHS and non NHS patients, and allows for patients to buy products (e.g., toothbrush, paste, etc.) when they pay for the received services (such as periodontal therapy with plague removal and scaling, and dental surgery).

# Structural Relationships in Class Diagrams

# What Is an Association?

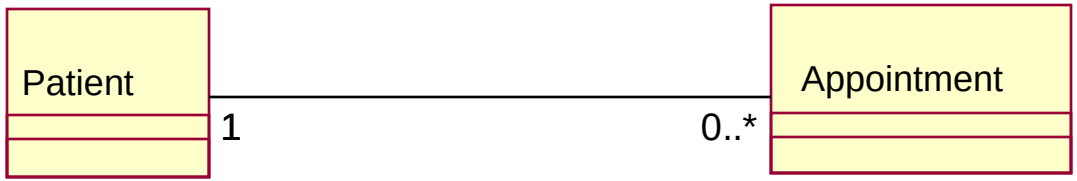
- The semantic relationship between two or more classifiers that specifies connections among their instances.

- A structural relationship specifying that objects of one thing are connected to objects of another thing.



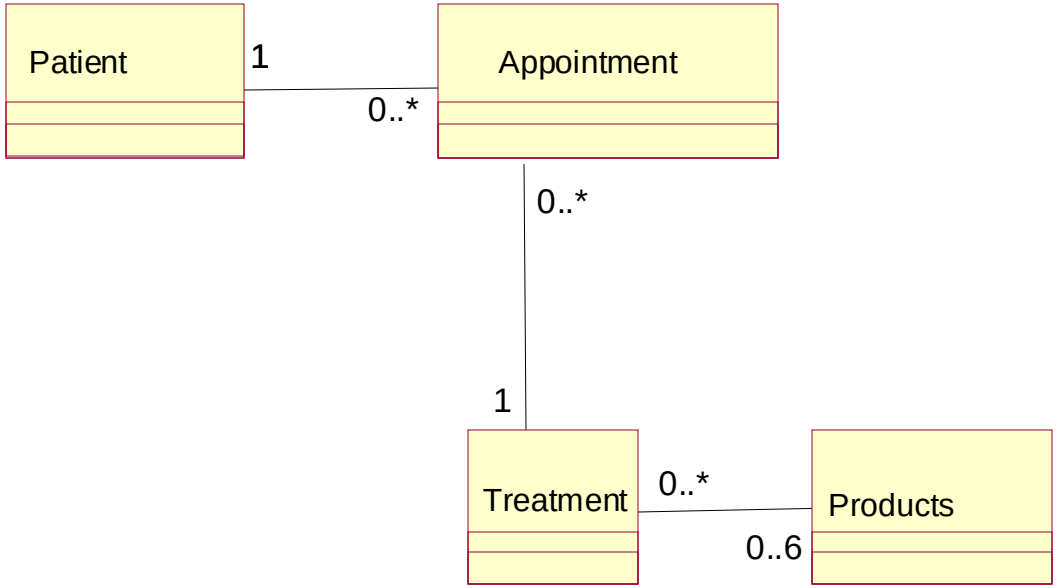
# What Is Multiplicity?

- Multiplicity is the number of instances one class relates to ONE instance of another class.
- For each association, there are two multiplicity decisions to make, one for each end of the association.
    - For each instance of Patient, many or no Appointments can be made.
    - For each instance of Appointment, there will be one Patient to see.



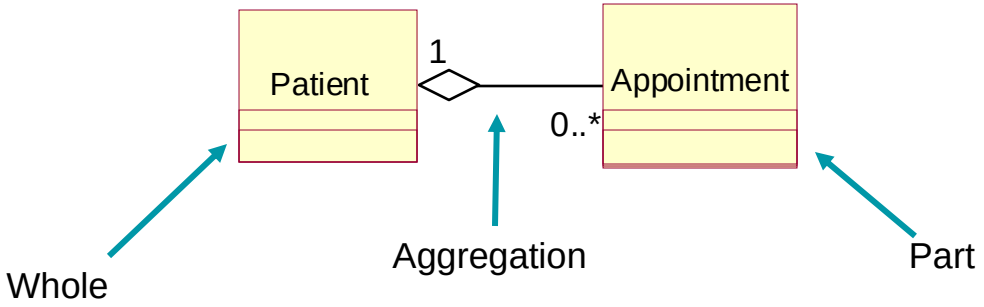
# Multiplicity: Example

| | |
|---|---|
| Unspecified | |
| Exactly One | 1 |
| Zero or More | 0..* |
| Zero or More | * |
| One or More | 1..* |
| Zero or One (optional value) | 0..1 |
| Specified Range | 2..4 |
| Multiple, Disjoint Ranges | 2, 4..6 |



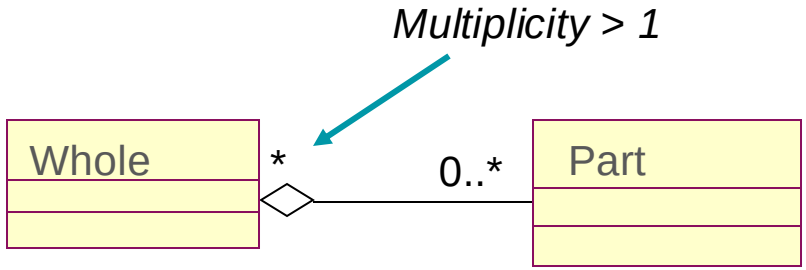
# What Is an Aggregation?

- A special form of association that models a whole-part relationship between the aggregate (the whole) and its parts.
  - An aggregation is an “is a part-of” relationship.
- Multiplicity is represented like other associations.

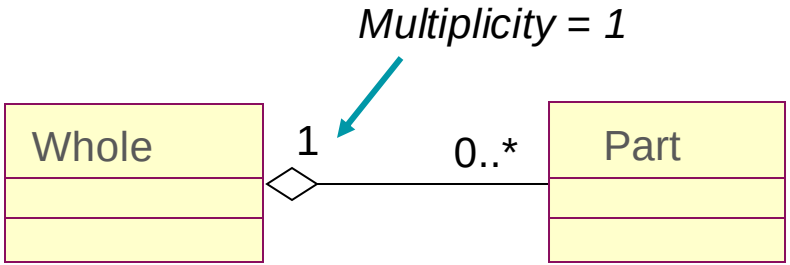


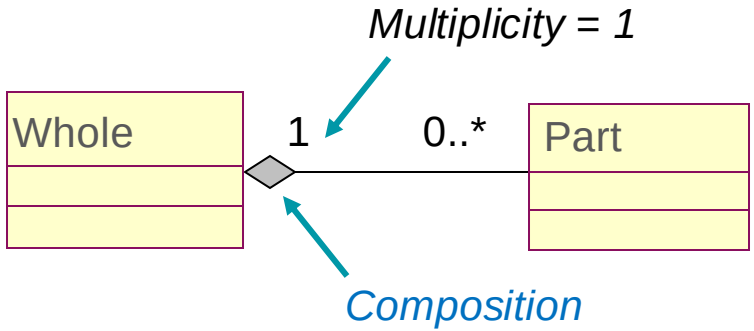
# Aggregation: Shared vs. Non-shared

- Shared Aggregation



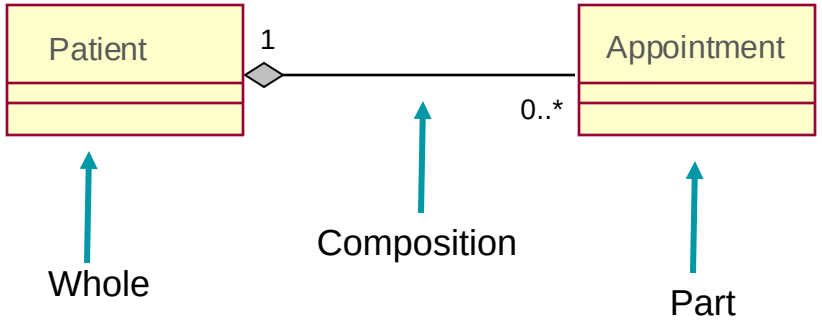

- Non-shared Aggregation



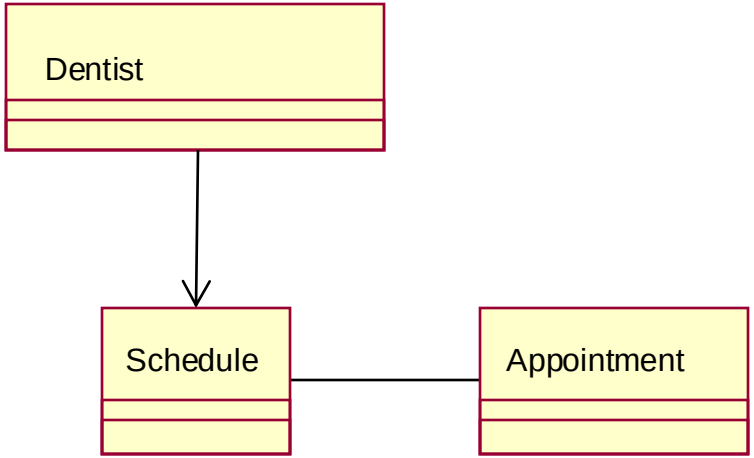
# What Is Composition?

- A form of aggregation with strong ownership and coincident lifetimes
  - The parts cannot survive the whole/aggregate



# What Is Navigability?

- Indicates that it is possible to navigate from an associating class to the target class using the association



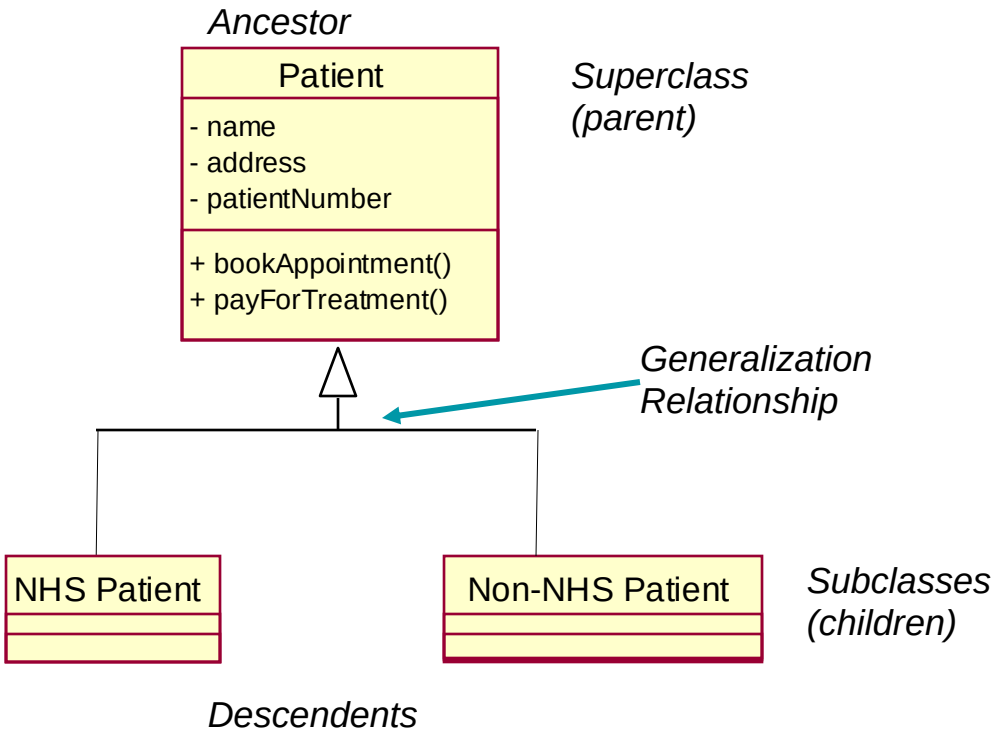
# What Is Generalization?

- A relationship among classes where one class shares the *properties and/or behavior* of one or more classes.
- Defines a hierarchy of abstractions where a subclass inherits from one or more superclasses.

- Is an "**is a kind of**" relationship.

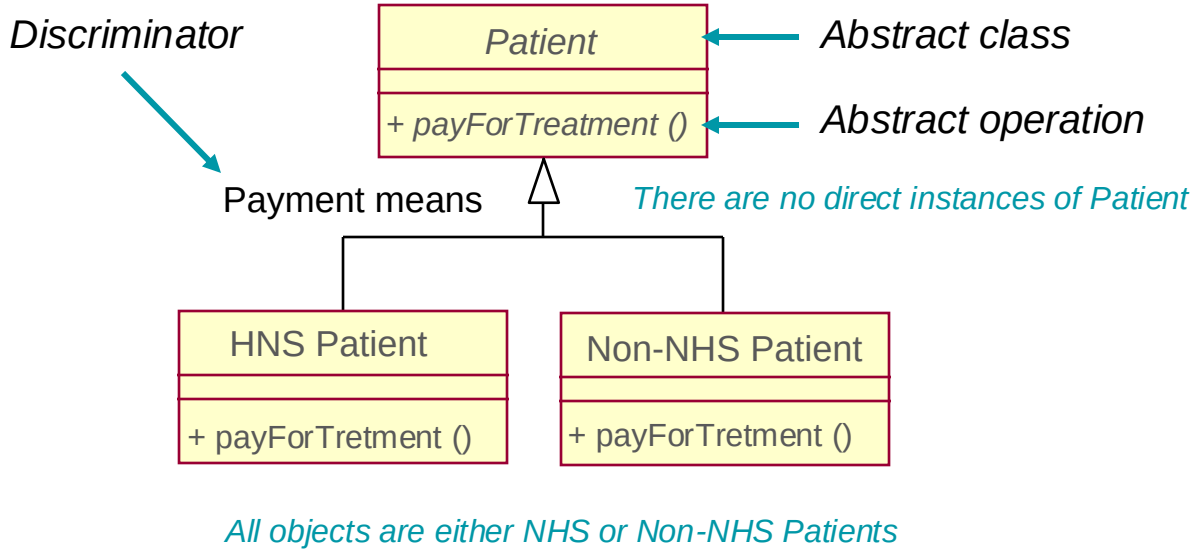
# Example: Inheritance

- One class inherits from another
- Follows the “is a” style of programming
- Class substitutability



# Abstract and Concrete Classes

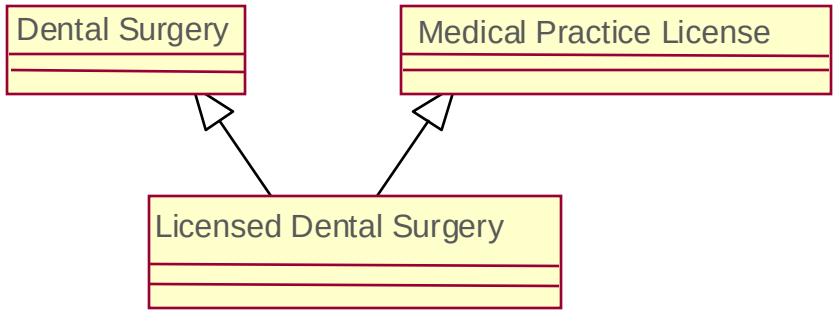
- Abstract classes cannot have any objects
- Concrete classes can have objects



# Generalization vs. Aggregation

- Generalization and aggregation are often confused
  - Generalization represents an “is a” or “kind-of” relationship
  - Aggregation represents a “part-of” relationship

Is this correct?



# UML: Sequence Diagrams

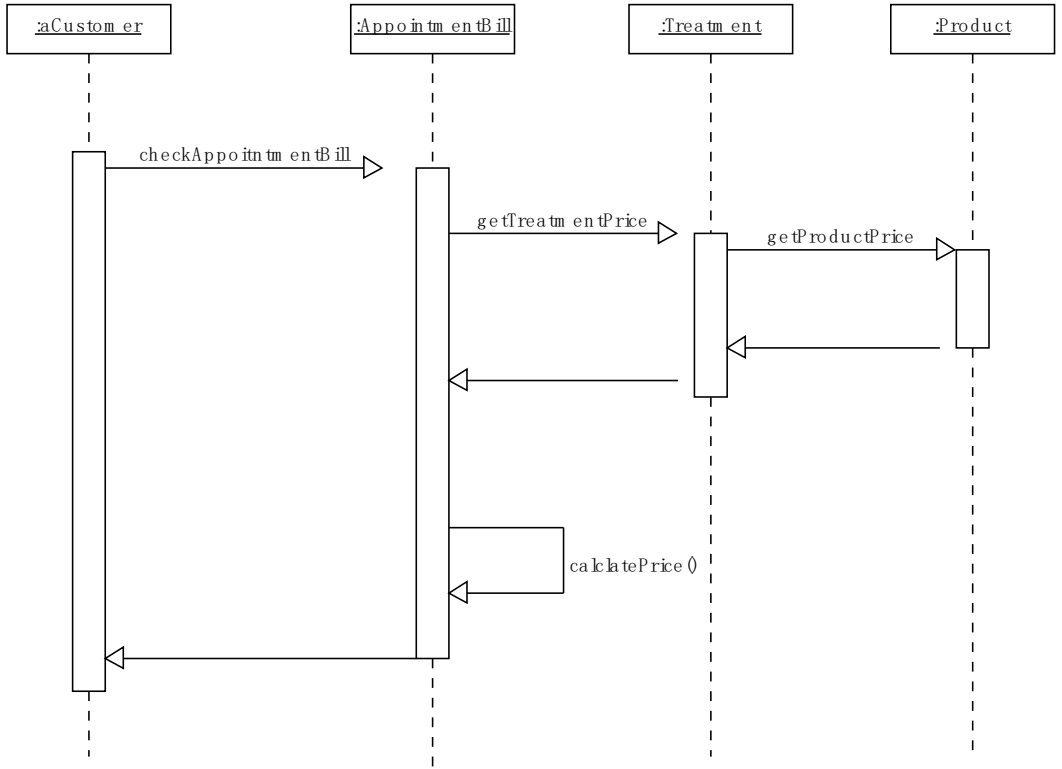
# Objects Need to Collaborate

- Objects are useless unless they can collaborate to solve a problem.
  - Each object is responsible for its own behavior and status.
  - No one object can carry out every responsibility on its own.

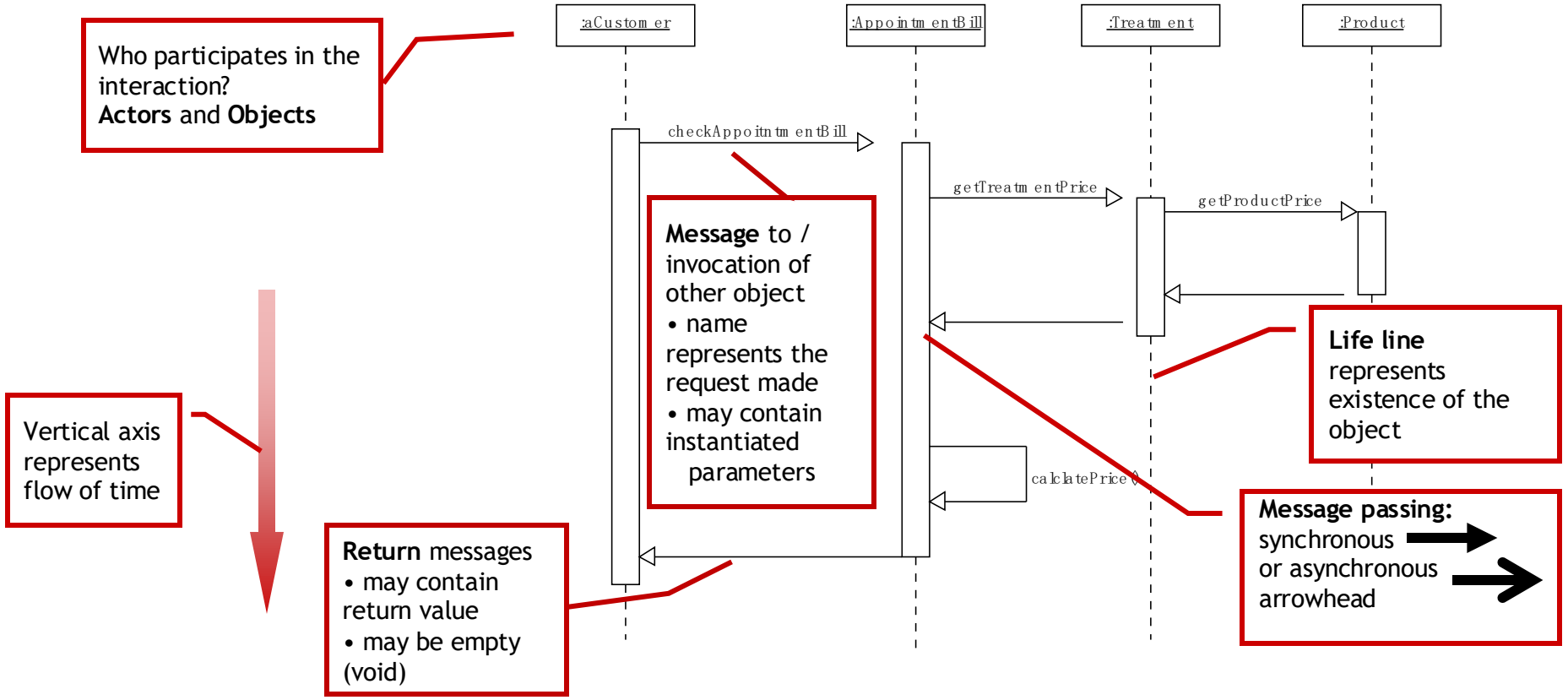
- How do objects interact with each other?
  - They interact through messages.
  - Message shows how one object asks another object to perform some activity.

# Sequence Diagrams: Basic Elements

- A set of participants arranged in time sequence
- Good for real-time specifications and complex scenarios



# Sequence Diagrams: Basic Elements



# Method for Analysis Sequence Diagrams

- for each scenario (high-level sequence diagram)
  - decompose to show what happens to objects inside the system

    ➔ objects and messages
  - Which tasks (operation) does the object perform?

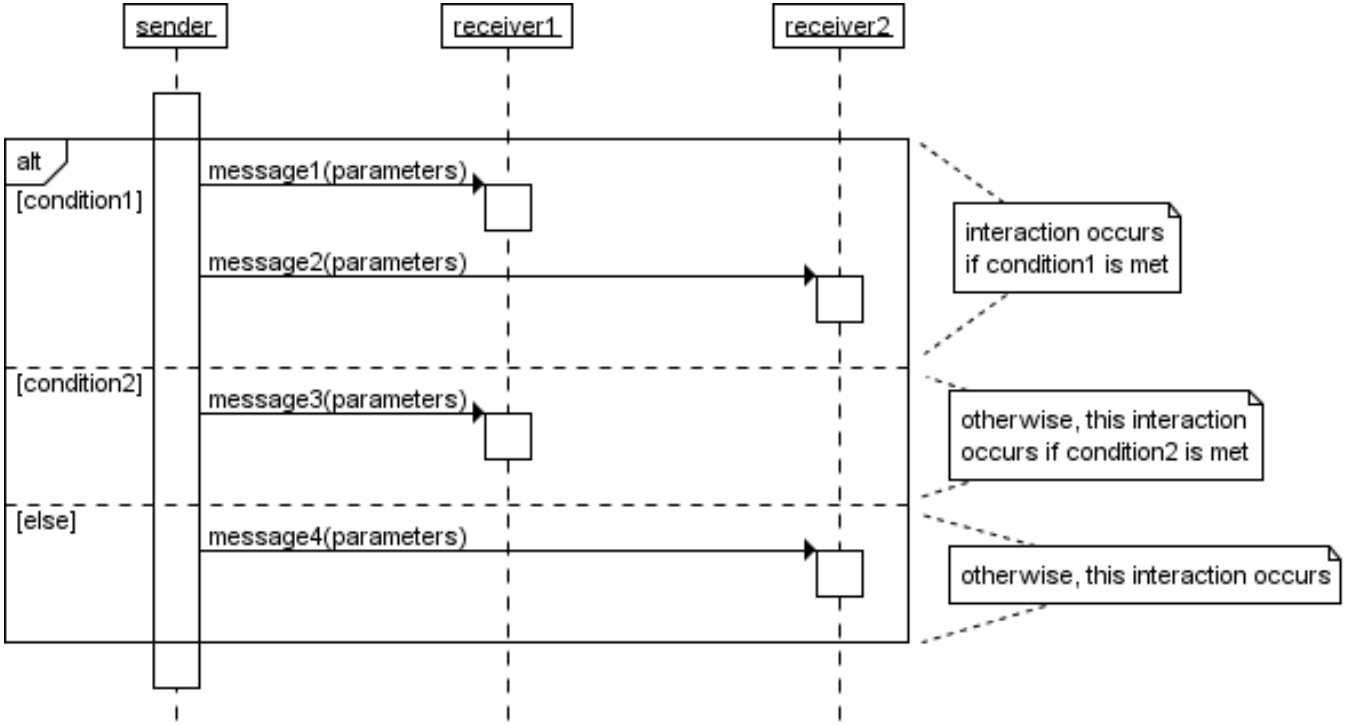
    ➔ label of message arrow
  - Who is to trigger the next step?

    ➔ return message or pass on control flow

# Sequence Diagrams

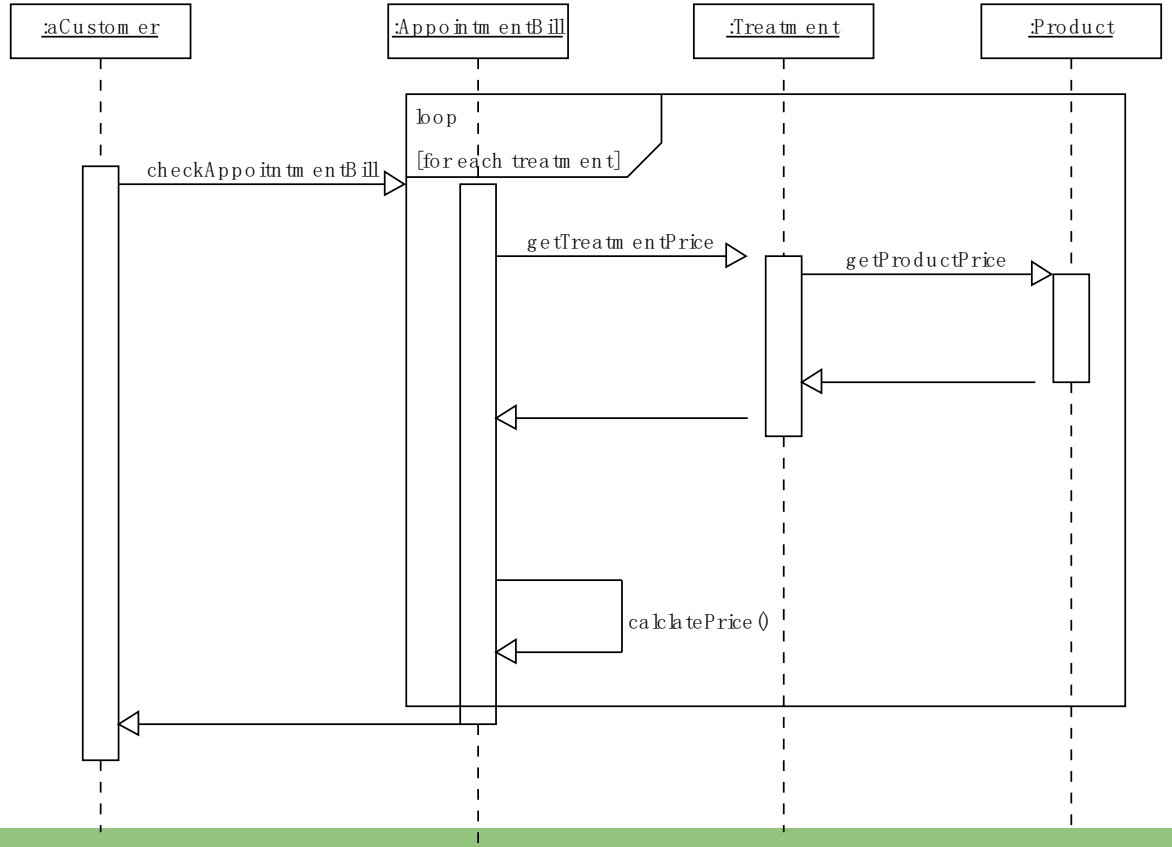
- Sequence Diagrams can model simple sequential flow, branching, iteration, recursion and concurrency
- They may specify different scenarios/runs
    - Primary
    - Variant
    - Exceptions

# Interaction frames: alt




https://web.archive.org/web/20231018070441/http://www.tracemodeler.com/articles/a_quick_introduction_to_uml_sequence_diagrams/

# Interaction frames: loop



# Review

- What are the OO Design Principles?
- What does a class diagram represent?
- Define association, aggregation, and generalization.
- How do you find associations?
- What information does multiplicity provide?
- What is the main purpose of a SD?
- What are the main concepts in a SD?