ECE444: Software Engineering

UML, OOP, Design Pattern 1

Shurui Zhou

The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

Learning Goals

- Understand UML
- Understand OOP
- Understand what drives design
- Understand information hiding

Introduction to Software Design

Goal of Software Design

- For each desired program behavior there are infinitely many programs that have this behavior
  - What are the differences between the variants?
  - Which variant should we choose?
- Since we usually have to synthesize rather than choose the solution…
  - How can we design a variant that has the desired properties?

A typical Intro of CS design process

1. Discuss software that needs to be written
2. Write some code
3. Test the code to identify the defects
4. Debug to find causes of defects
5. Fix the defects
6. If not done, return to step 1

A Better Software Design

- Think before coding: broadly consider quality attributes
  - Maintainability, extensibility, performance, …
- Propose, consider design alternatives
  - Make explicit design decision

Using a Design Process

- A design process organizes your work
- A design process structures your understanding
- A design process facilitates communication

Why a Design Process?

- Without a process, how do you know what to do?
  - –A process tells you what is the next thing you should be doing
- A process structures learning
  - –We can discuss individual steps in isolation
  - –You can practice individual steps, too
- If you follow a process, we can help you better
  - –You can show us what steps you have done
  - –We can target our advice to where you are stuck

- **Design goals** enable evaluation of designs

– e.g. maintainability, reusability, scalability

- **Design principles** are heuristics that describe best practices

– e.g. high correspondence to real-world concepts

- **Design patterns** codify repeated experiences, common solutions

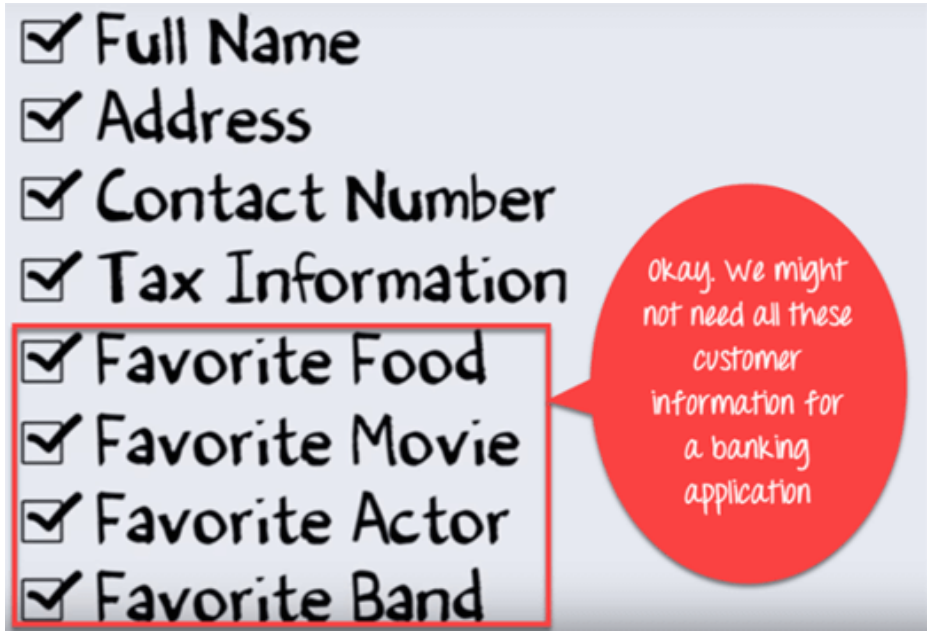
– e.g. template method pattern

Fundamental Object-Oriented Design Principle

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

OOP - Abstraction

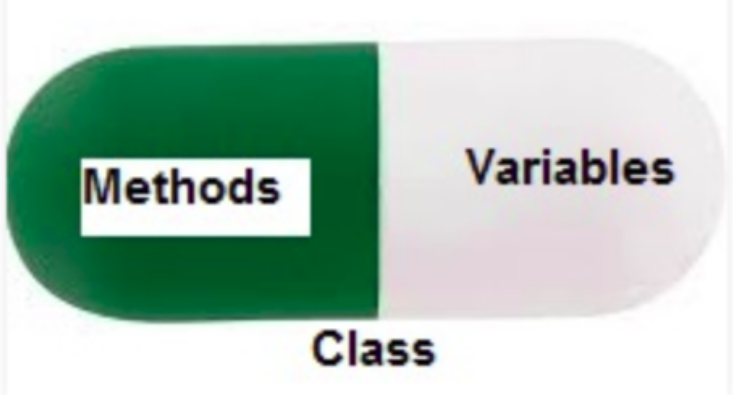
- "shows" only essential attributes and "hides" unnecessary information.
- Think about a banking application, you are asked to collect all the information about your customer.



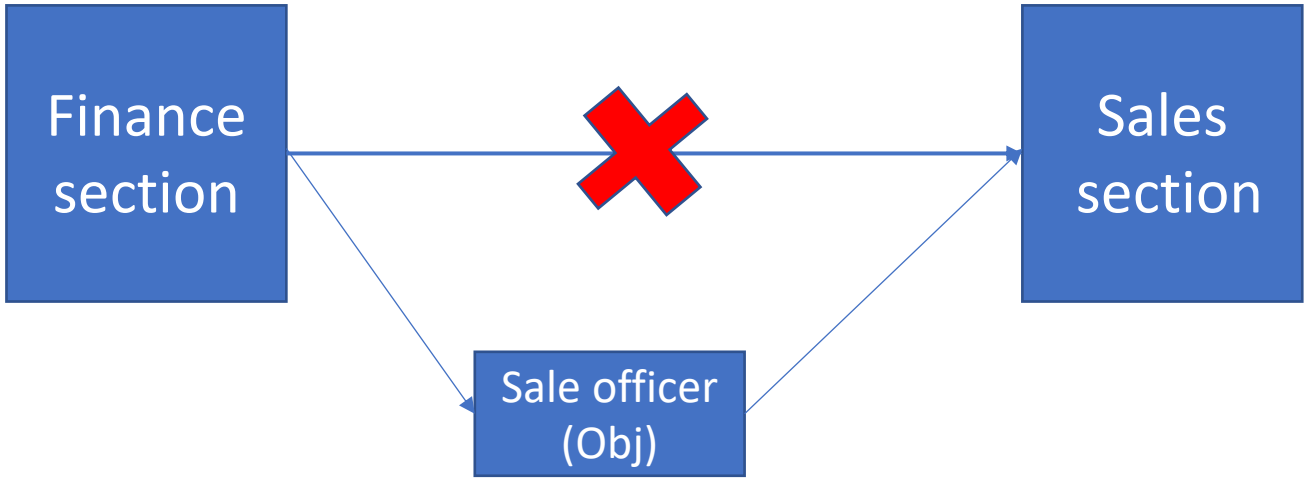
Fundamental Object-Oriented Design Principle

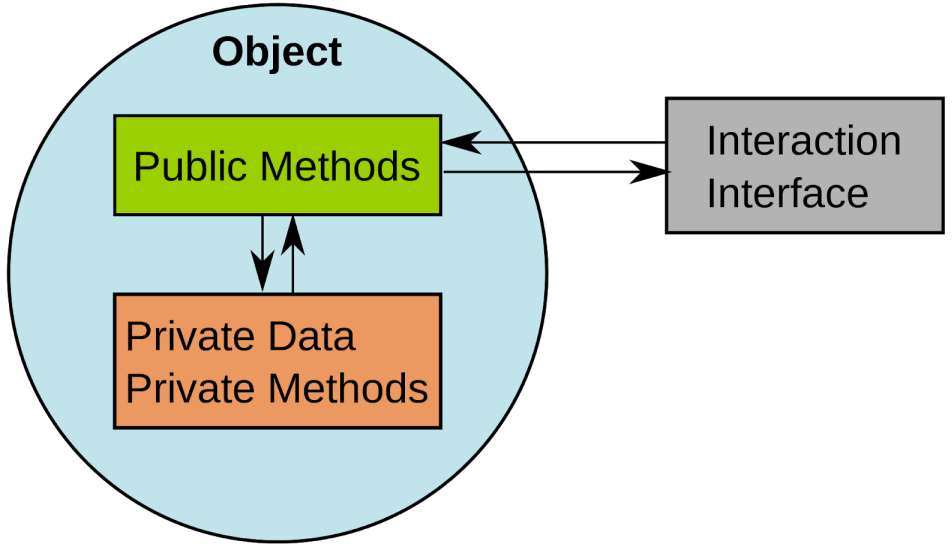
- **Abstraction**
- **Encapsulation** bundling data and methods that work on that data within one unit, e.g., a class in Java.
- Modularity
- Hierarchy

OOP - Encapsulation



- A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.
- Consider a real-life example, in a company:





Fundamental Object-Oriented Design Principle

Difference between Abstraction and Encapsulation

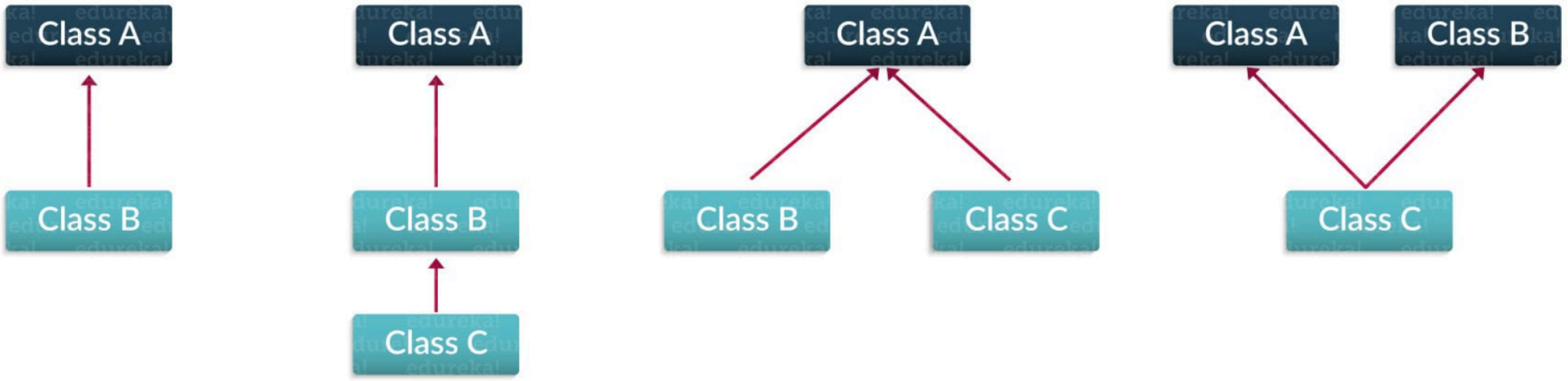
Abstraction	Encapsulation
Abstraction solves the issues at the design level.	Encapsulation solves it implementation level.
Abstraction is about hiding unwanted details while showing most essential information.	Encapsulation means binding the code and data into a single unit.
Abstraction allows focussing on what the information object must contain	Encapsulation means hiding the internal details or mechanics of how an object does something for security reasons.

Fundamental Object-Oriented Design Principle

- **Abstraction** "shows" only essential attributes and "hides" unnecessary information.
- **Encapsulation** bundling data and methods that work on that data within one unit, e.g., a class in Java.
- **Inheritance** inheriting or transfer of characteristics from parent to child class without any modification”
- Polymorphism

Types Of Inheritance

edureka!



Single Inheritance

Multilevel Inheritance

Hierarchical Inheritance

Multiple Inheritance

Fundamental Object-Oriented Design Principle

- **Abstraction** "shows" only essential attributes and "hides" unnecessary information.
- **Encapsulation** bundling data and methods that work on that data within one unit, e.g., a class in Java.
- **Inheritance** inheriting or transfer of characteristics from parent to child class without any modification”
- **Polymorphism** a property of an object which allows it to take multiple forms.

OOP - Polymorphism

- a property of an object which allows it to take multiple forms.

```
4 # len() being used for a string
5 print(len("geeks"))
6
7 # len() being used for a list
8 print(len([10, 20, 30]))
```

Output:

```
5
3
```

```
4 def add(x, y, z = 0):
5     return x + y+z
6
7 # Driver code
8 print(add(2, 3))
9 print(add(2, 3, 4))
```

```
5
9
```

OOP

- **Abstraction** "shows" only essential attributes and "hides" unnecessary information.
- **Encapsulation** bundling data and methods that work on that data within one unit, e.g., a class in Java.
- **Inheritance** inheriting or transfer of characteristics from parent to child class without any modification”
- **Polymorphism** a property of an object which allows it to take multiple forms.

Modeling Notations

- Used for both requirements analysis and for specification and design
  - Useful for technical people
  - Provide a high-level view
  - Descendent of Entity-Relationship Diagrams
  - Describes data and operations
  - Require training
  - Many notations
    - each good for something
    - none good for everything

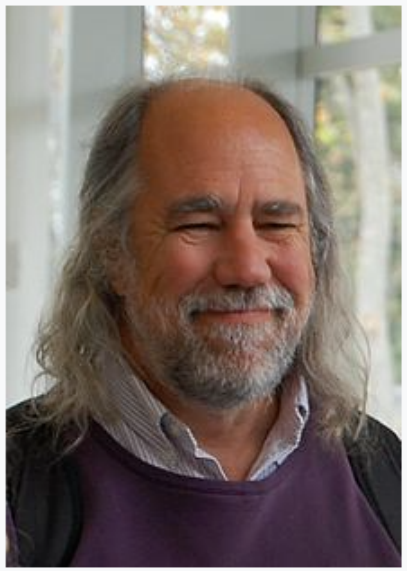
Origin of UML

Grady Booch Diagrams +

Jim Rumbaugh (OMT) Object Diagrams +

Ivar Jacobson use case diagrams

Three Amigos

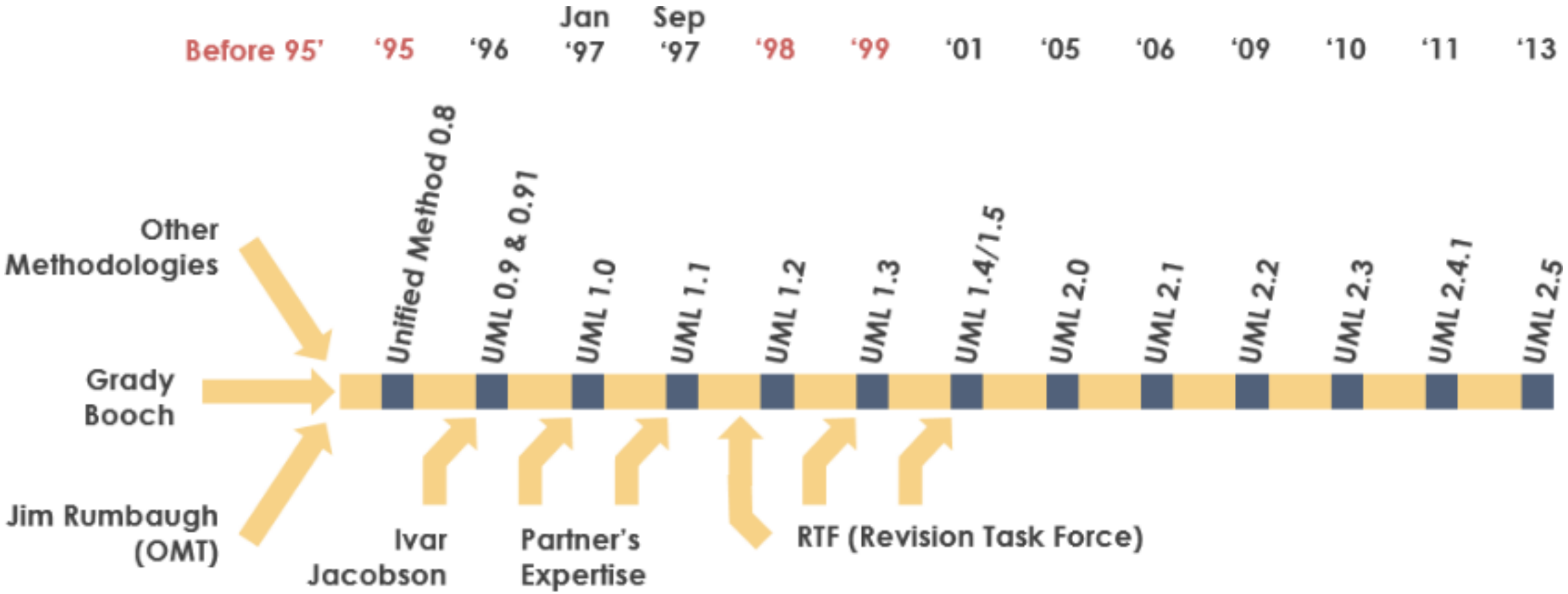


Grady Booch in 2011


Jim Rumbaugh


Ivar Jacobson

UML Timeline



Before 95' - Fragmentation ▶ 95' - Unification ▶ 98' - Standardization ▶ 99' - Industrialization

Usage of UML

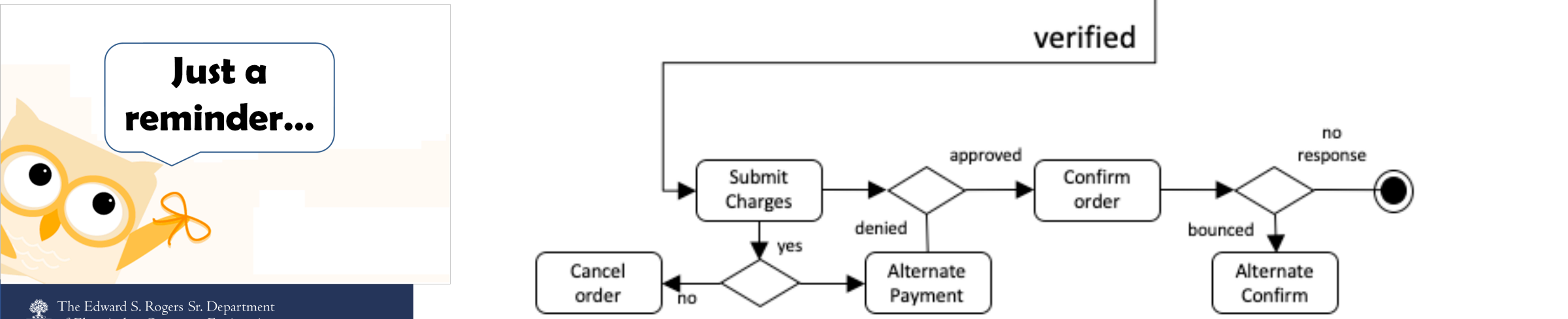
- Help developers communicate
- Provide documentation
- Help find errors (tools check for consistency)
- Generate code (with tools)
- Drawing Tools: ArgoUML, Visio (Microsoft), OmniGraffle

Types of UML

- **Behavioral UML Diagram**
- **Structural UML Diagram**

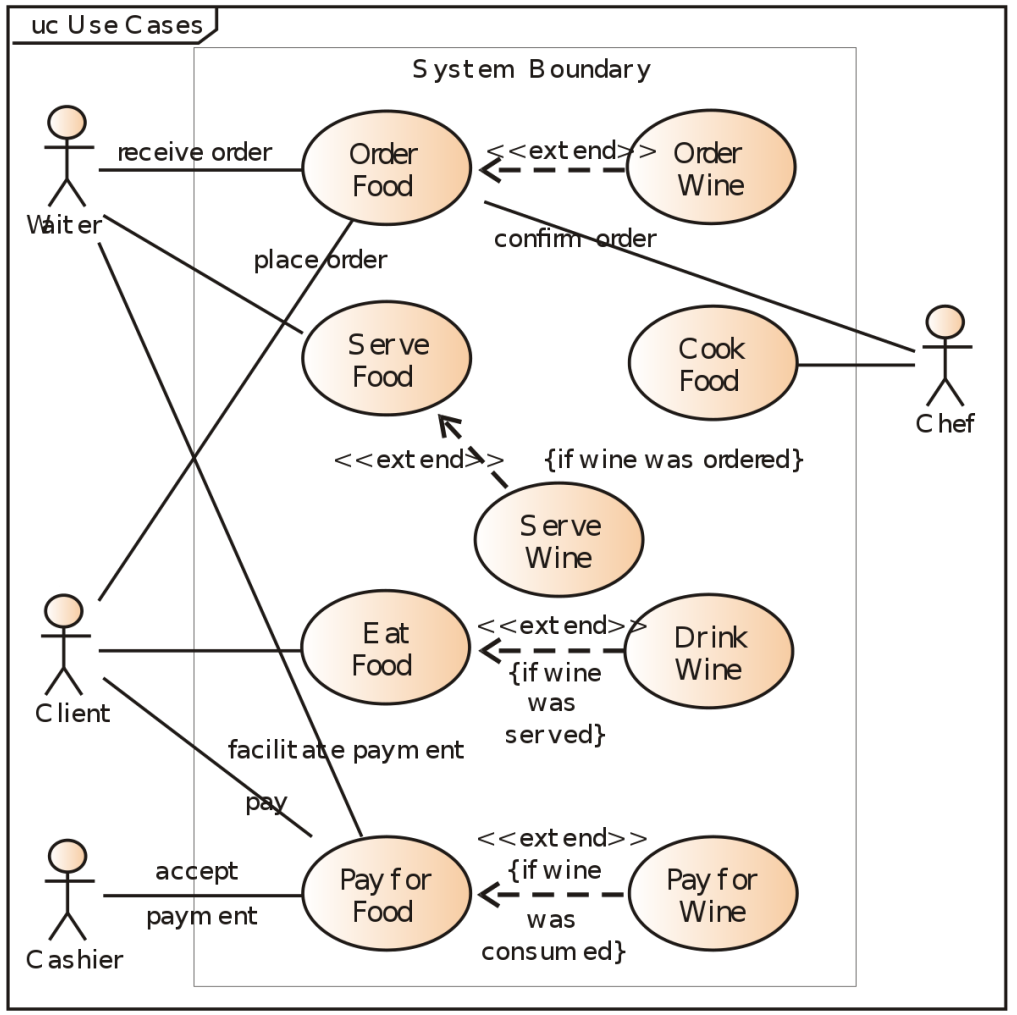
Behavioral UML Diagram - Activity Diagram

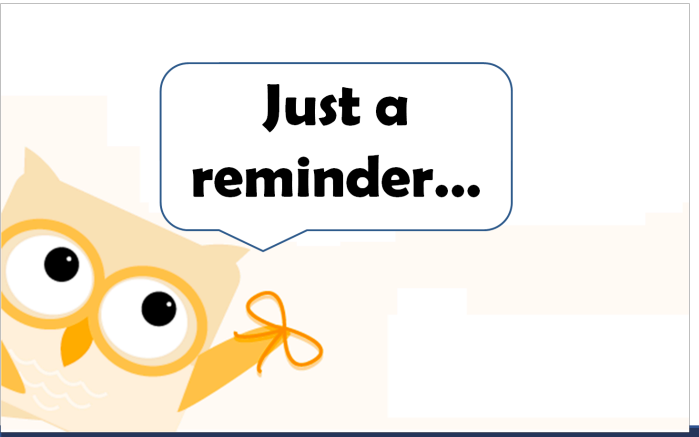
- The dynamic nature of a system by forming the flow of control from activity to activity



Behavioral UML Diagram – Use Case Diagram

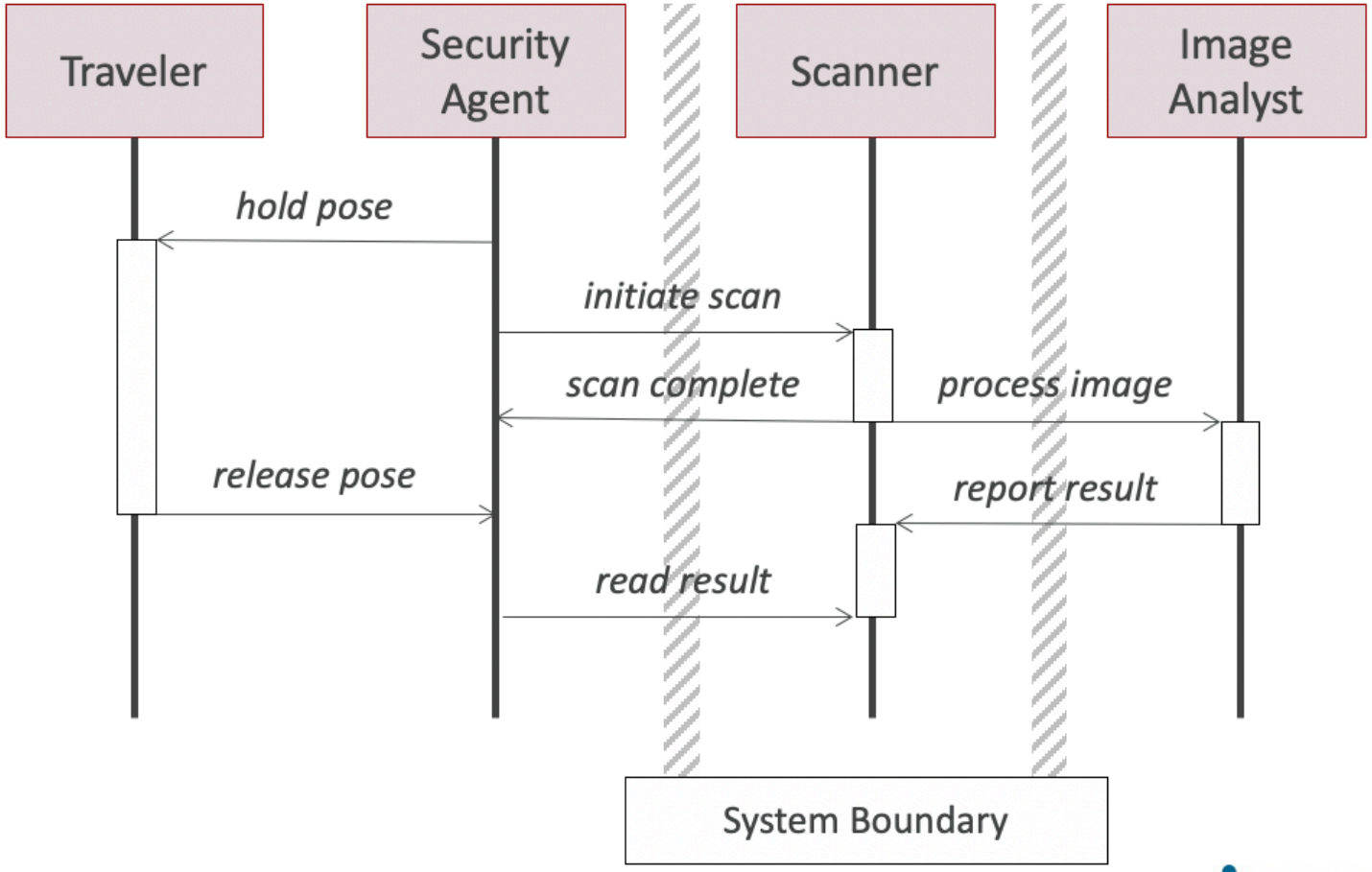
- Actor + Action

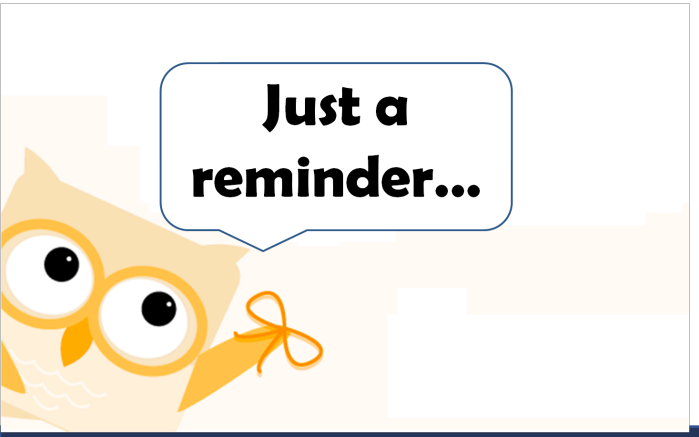




Behavioral UML Diagram - Sequence Diagram

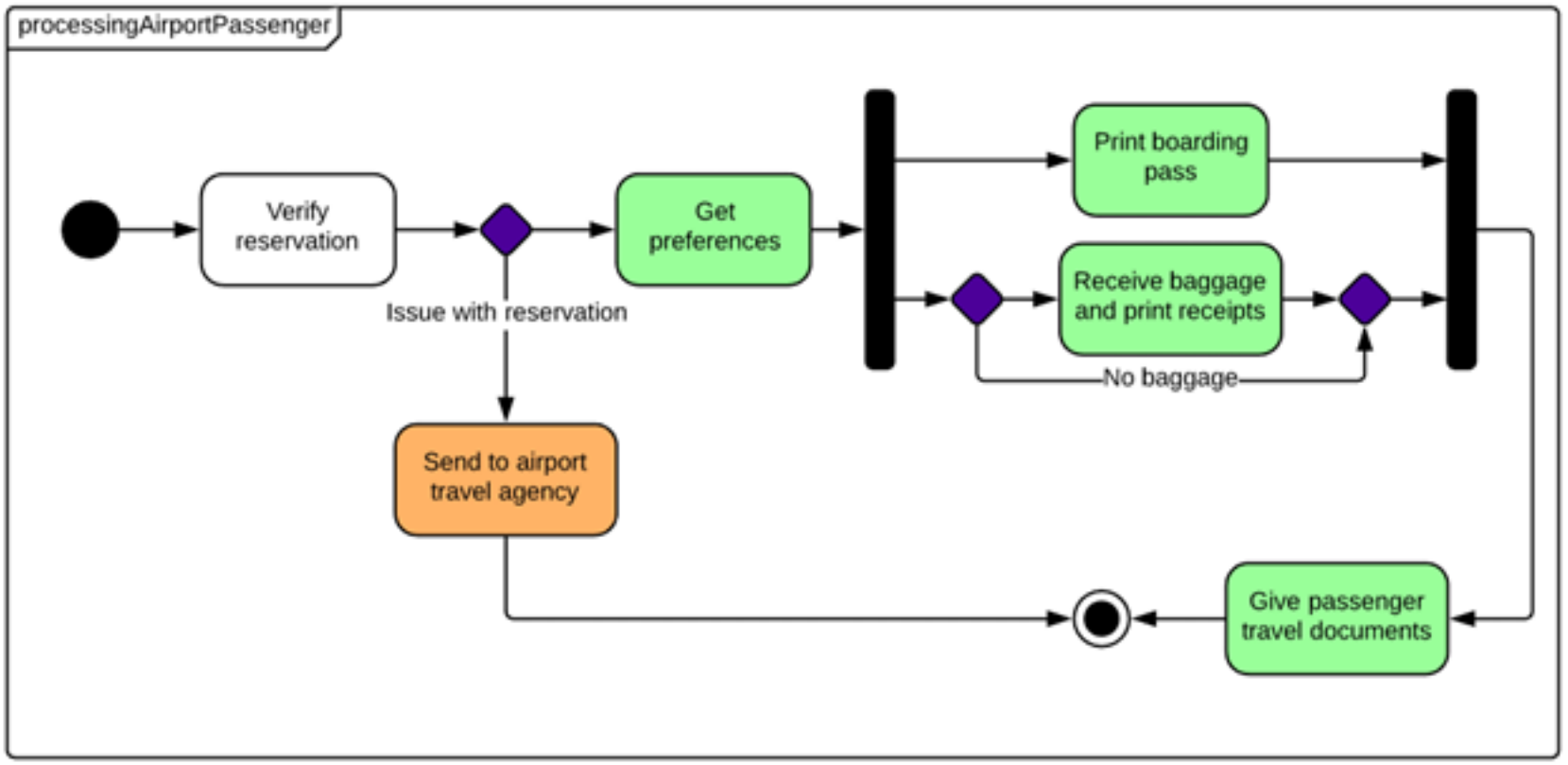
- The time sequence of the objects participating in the interaction





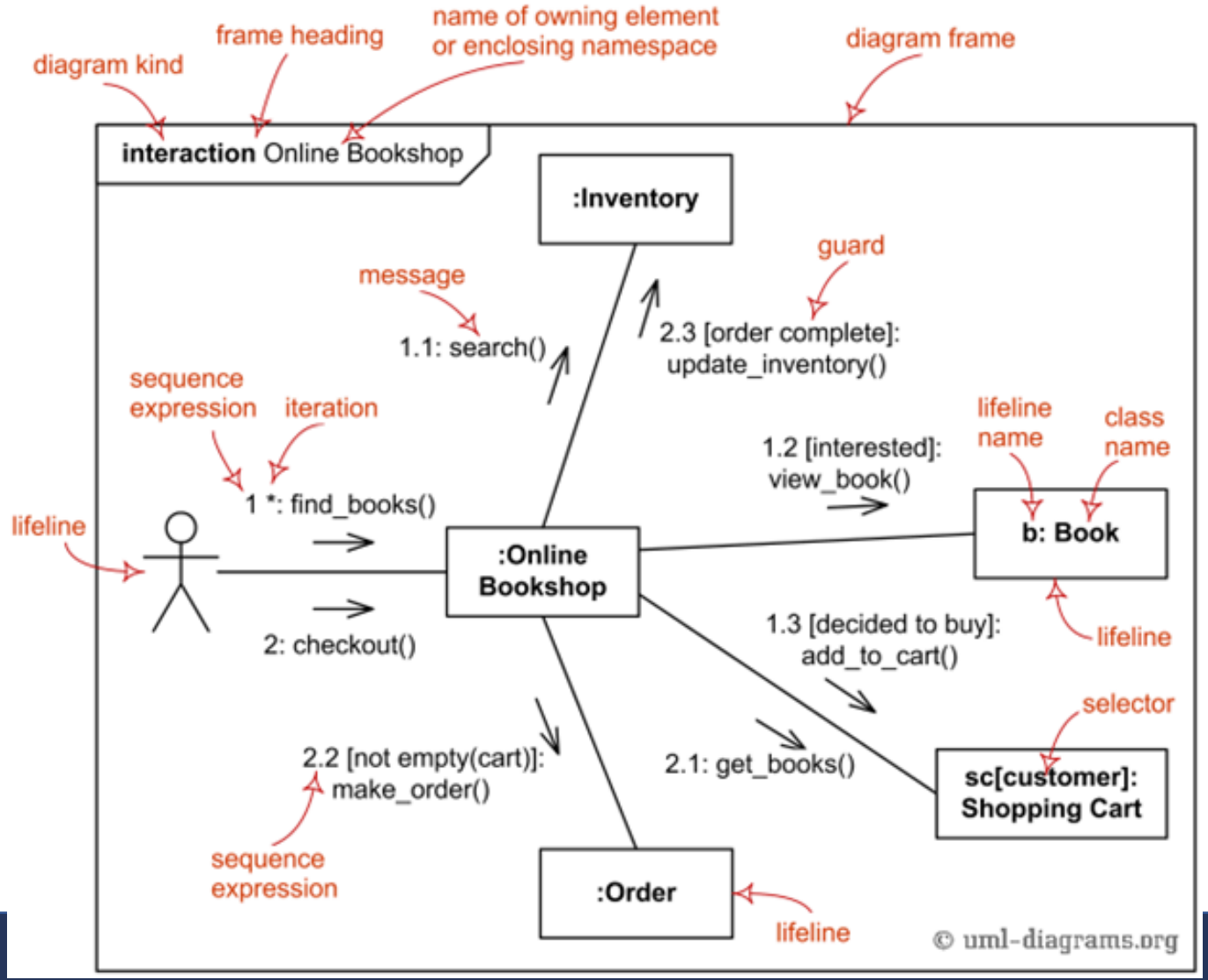
Behavioral UML Diagram – State Diagram

- possible states that an object of interaction goes through when an event occurs.



Behavioral UML Diagram – Communication Diagram

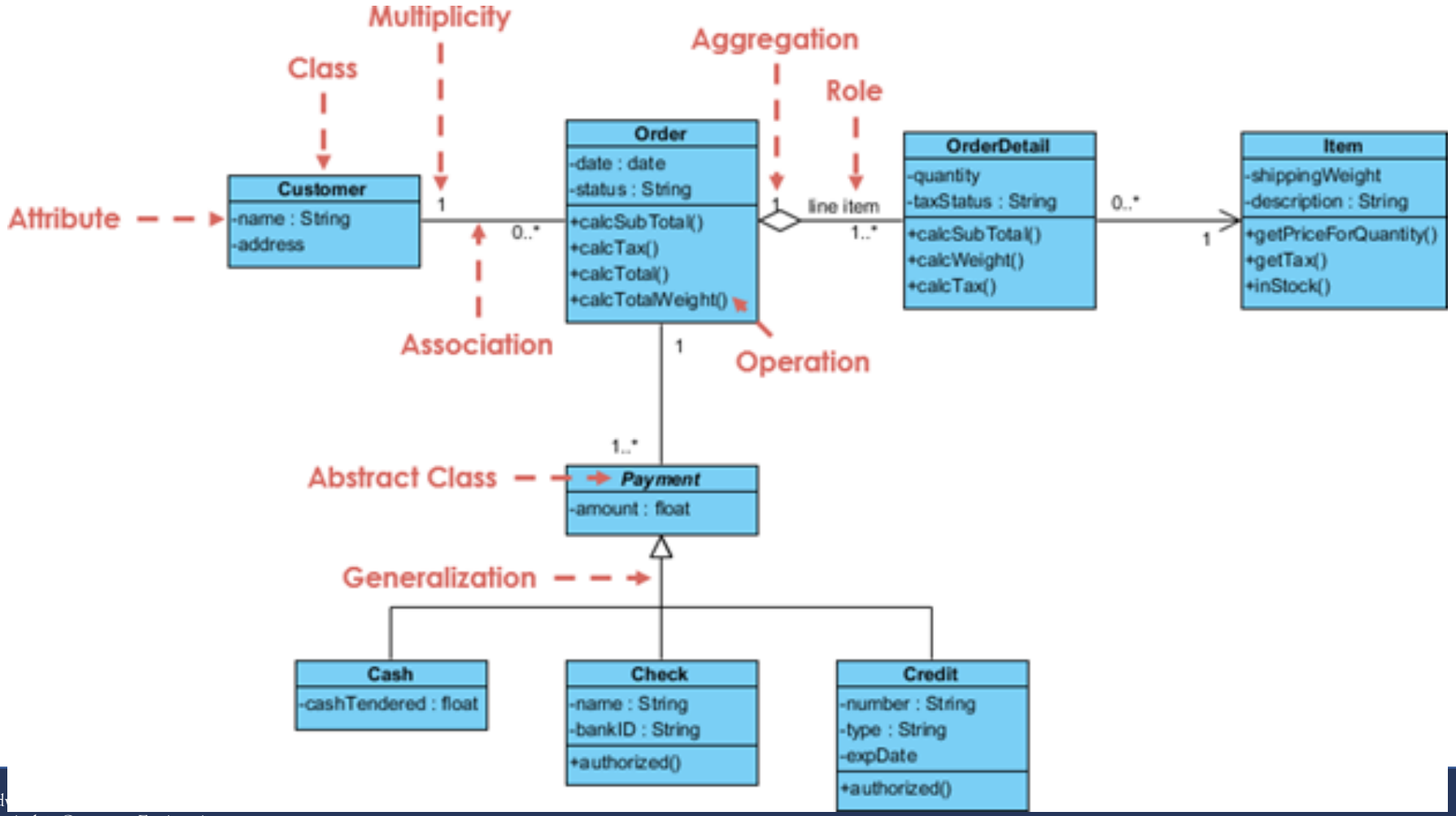
- Focus on the messages that are exchanged between the objects.



Types of UML

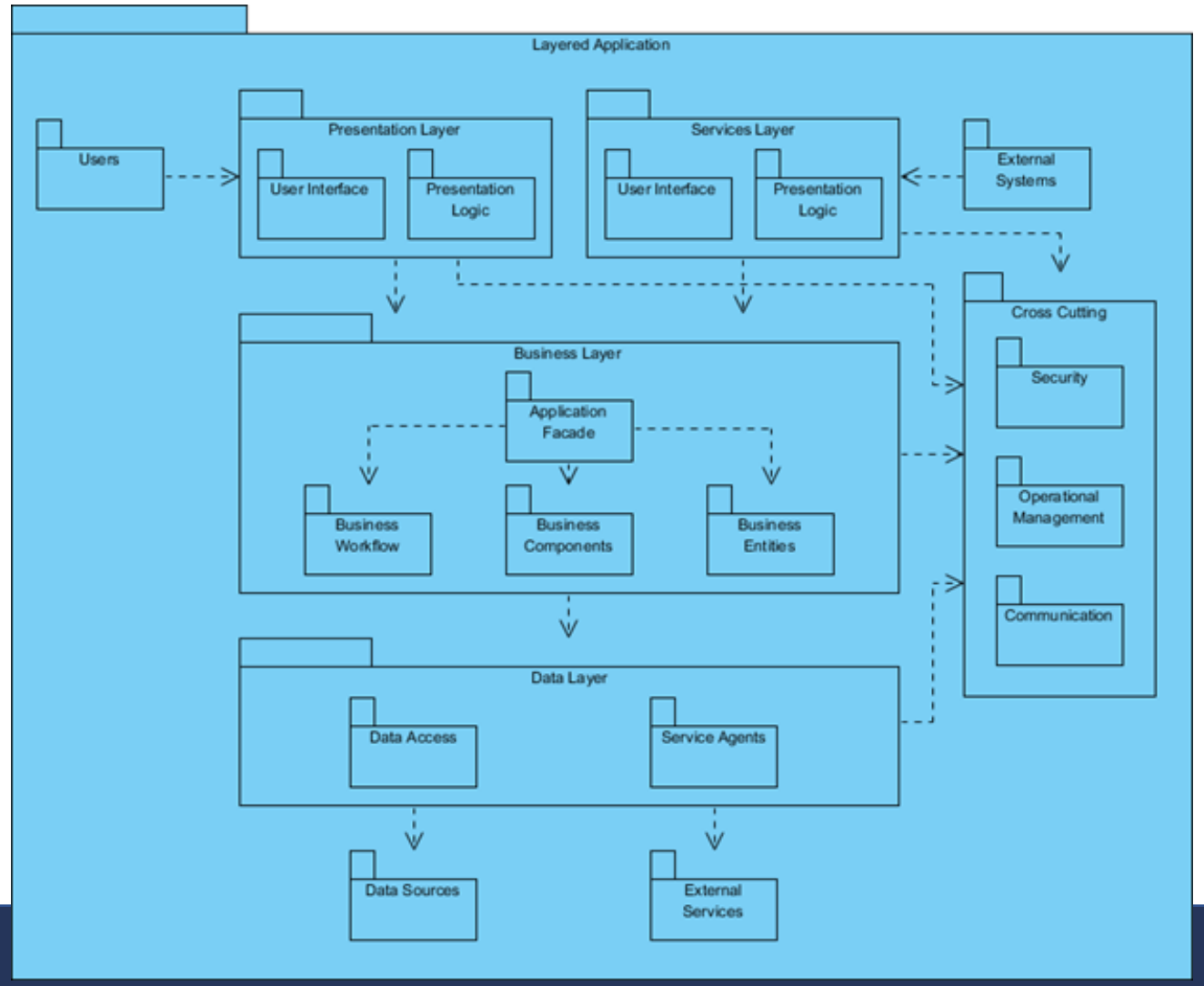
- **Behavioral UML Diagram**
- **Structural UML Diagram**

Structural UML Diagram - Class Diagram



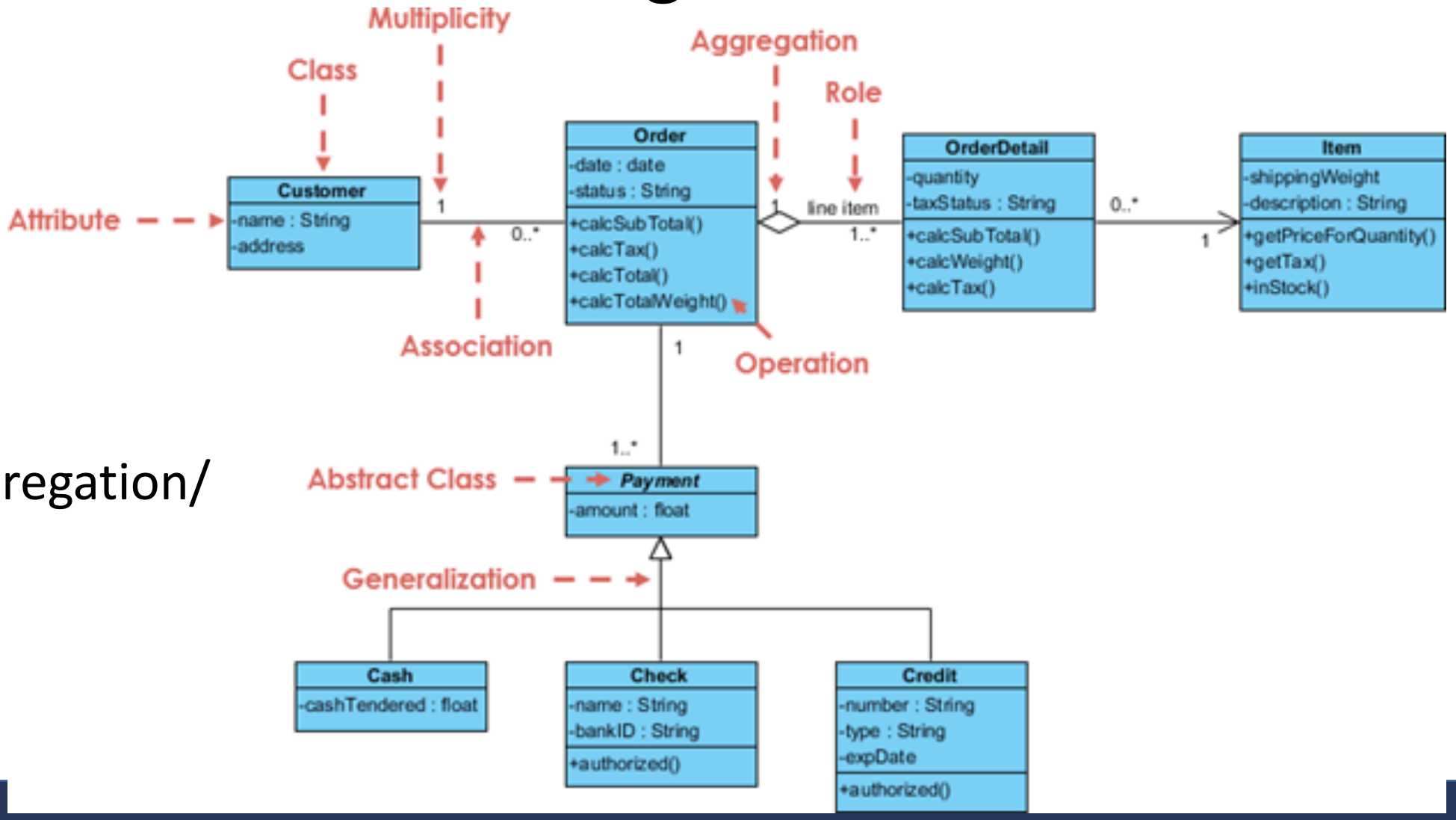
Structural UML Diagram - Package Diagram

- Package UML diagrams bring together the elements of a system into related groups to reduce dependencies between sets.



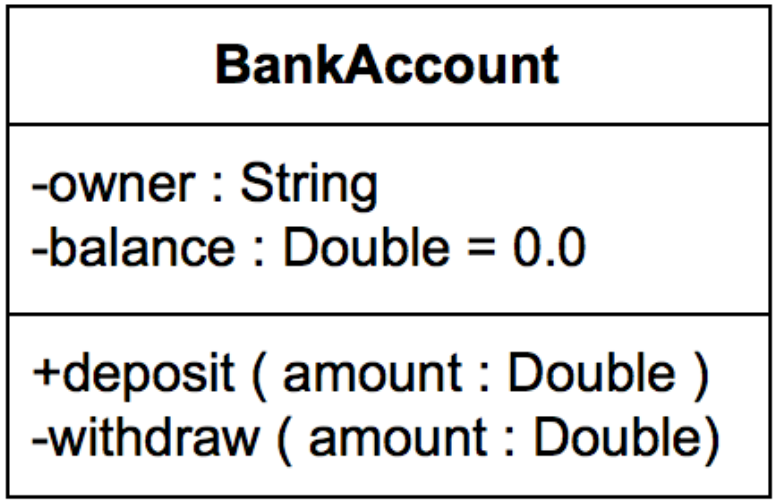
Elements of UML Class Diagram

- Class
  - attributes
  - operations
- Associations
  - multiplicity
  - direction/aggregation/
- Generalization



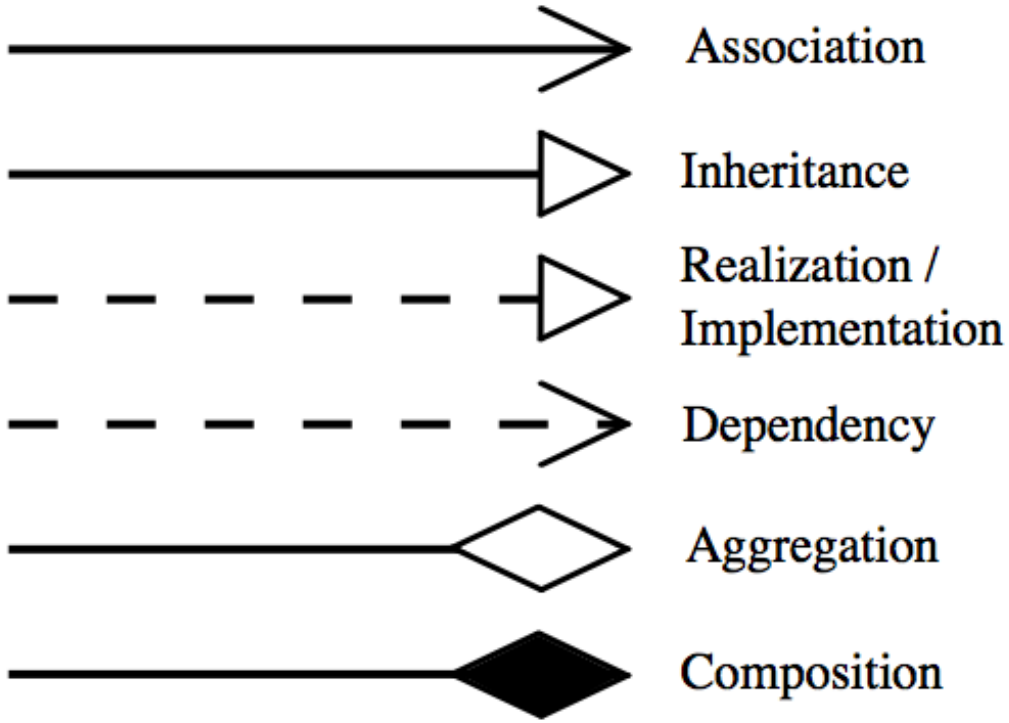
Class

- class name
- class attributes [attribute name : type]
- class methods [parameter: type]

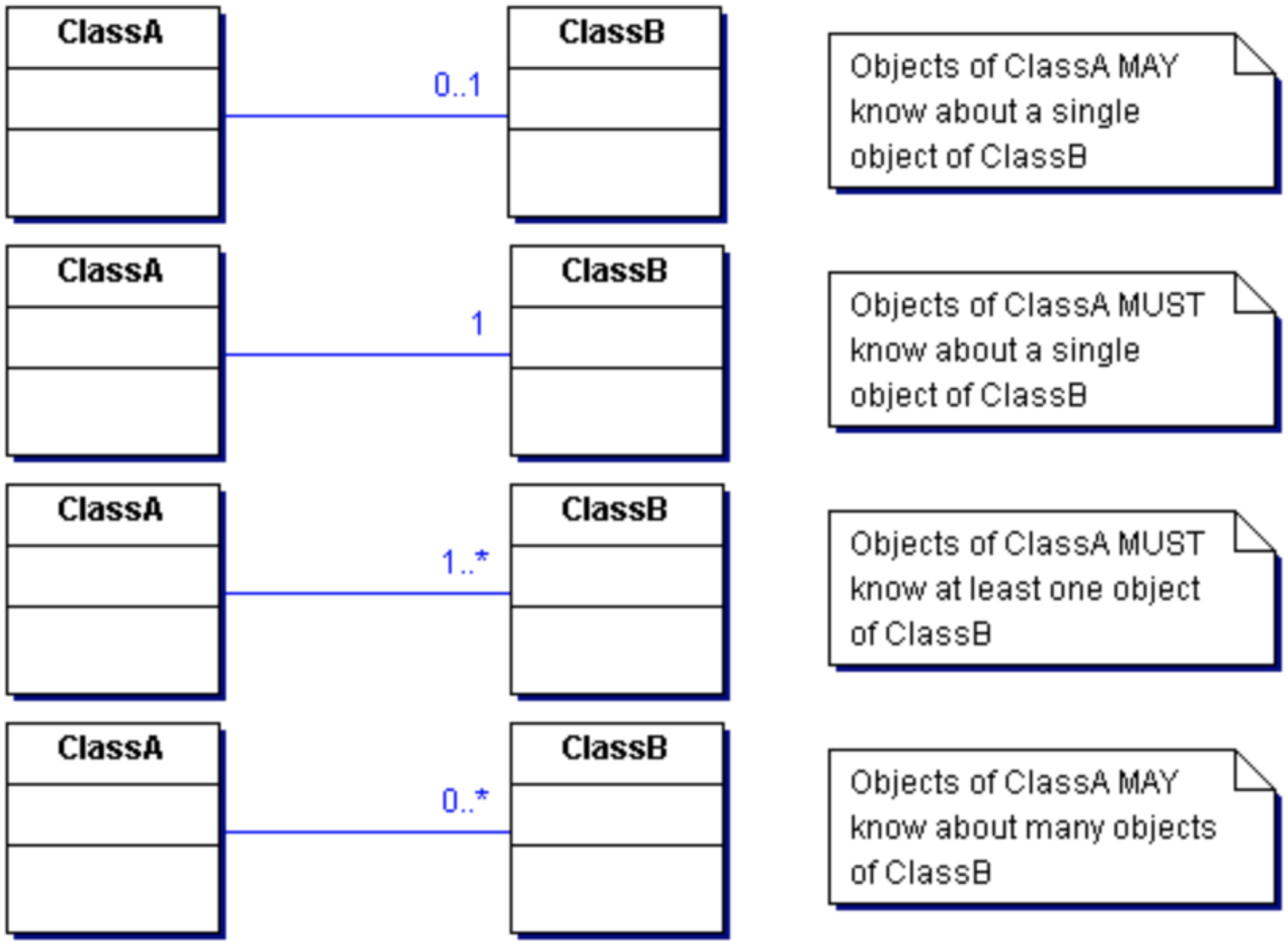


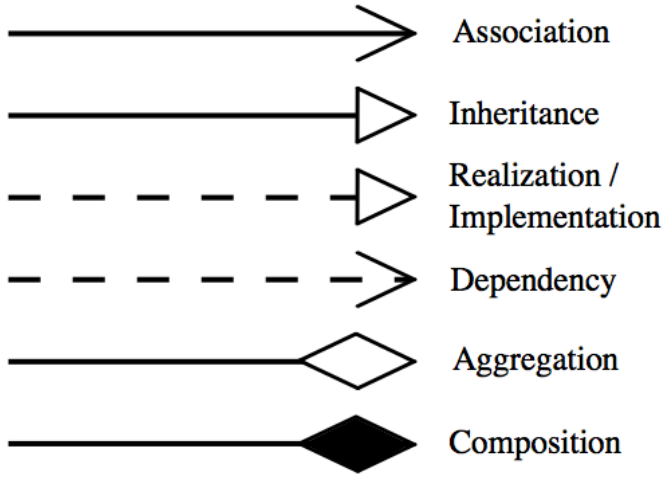
public	+	anywhere in the program and may be called by any object within the system
private	-	the class that defines it
protected	#	(a) the class that defines it or (b) a subclass of that class

Relationships



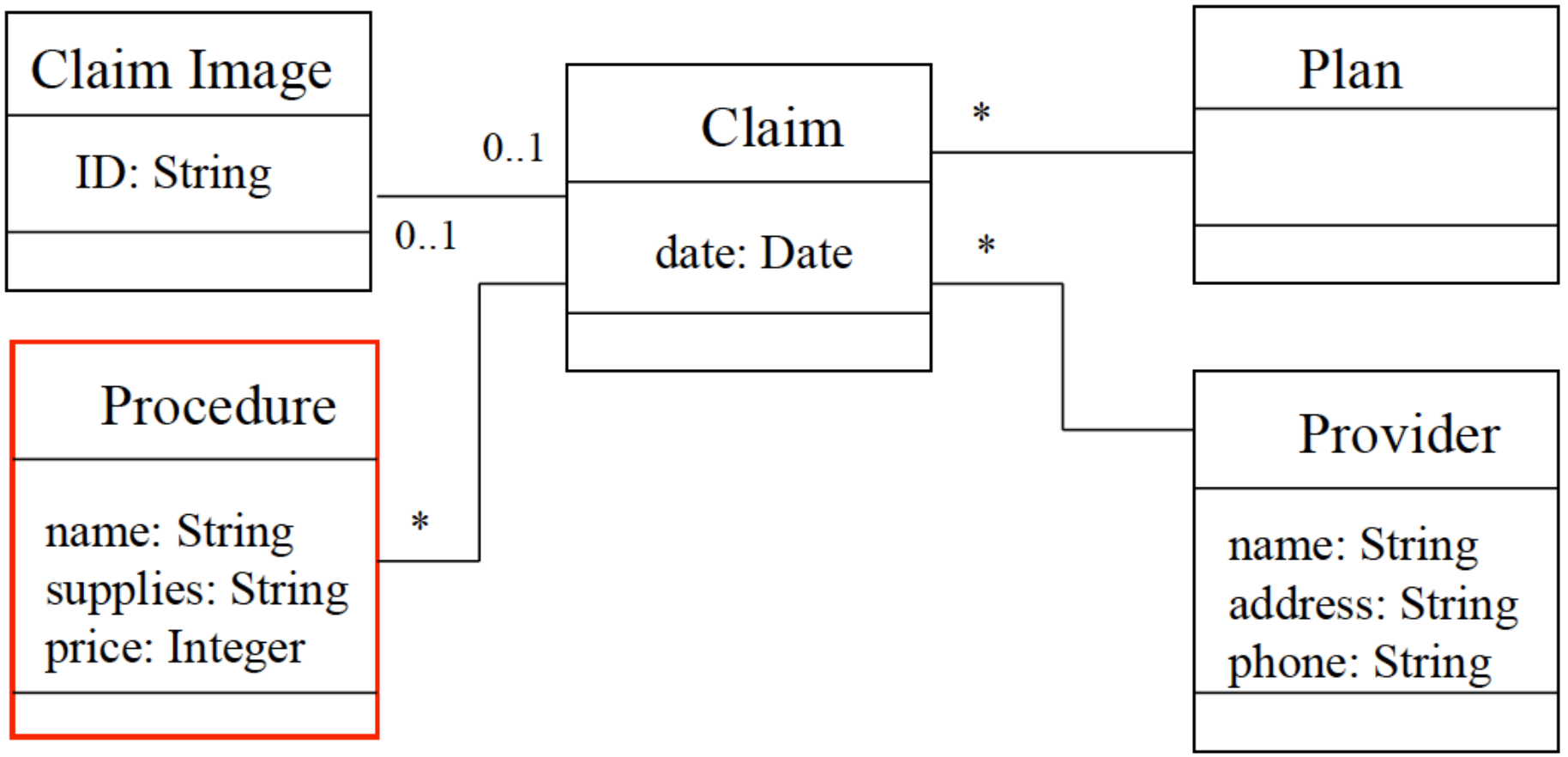
Association



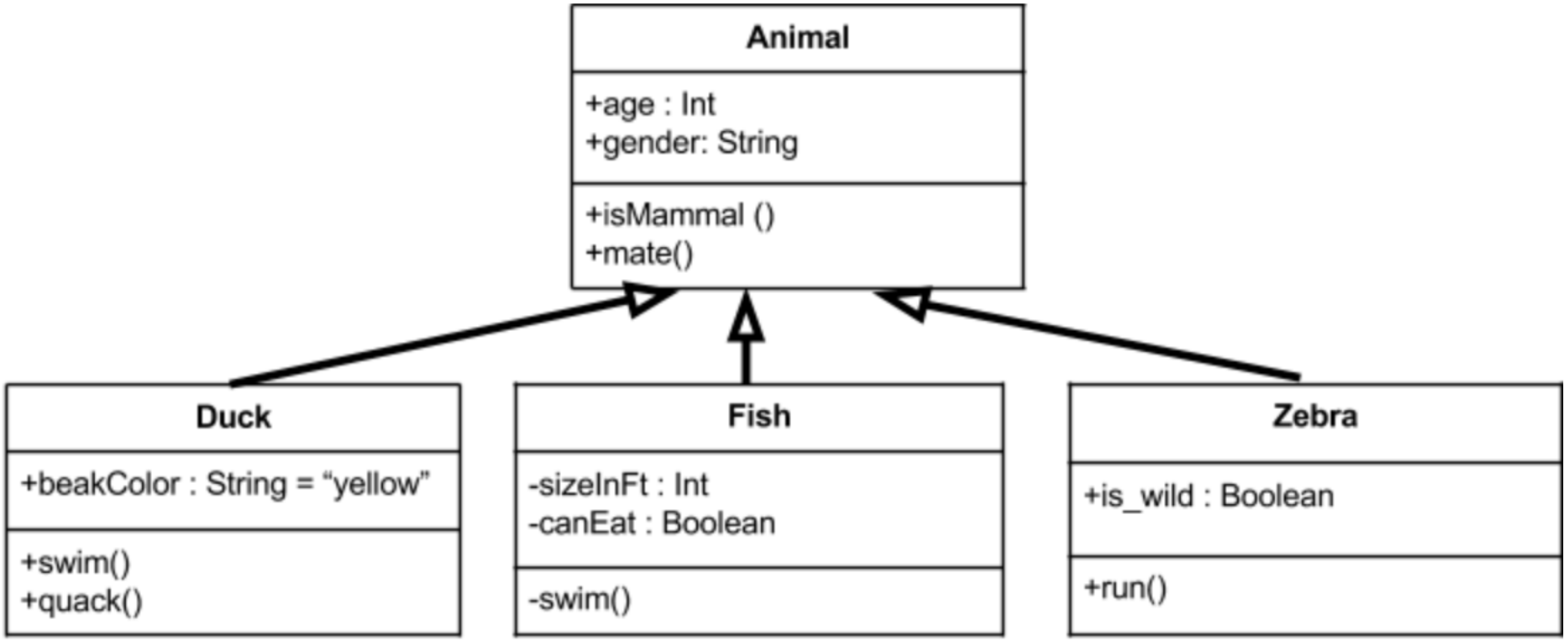


Multiplicity

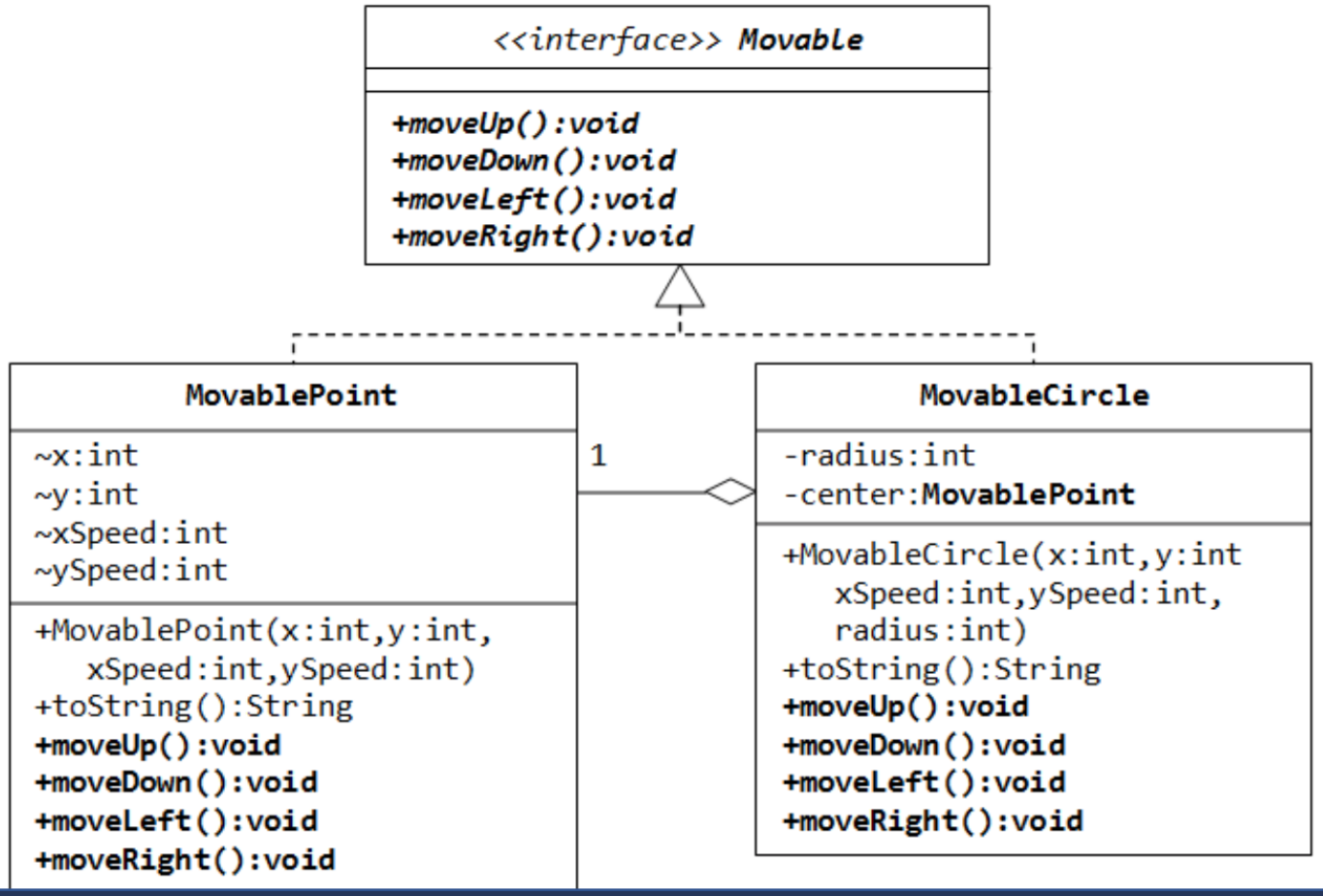
Attributes vs Associations



Inheritance



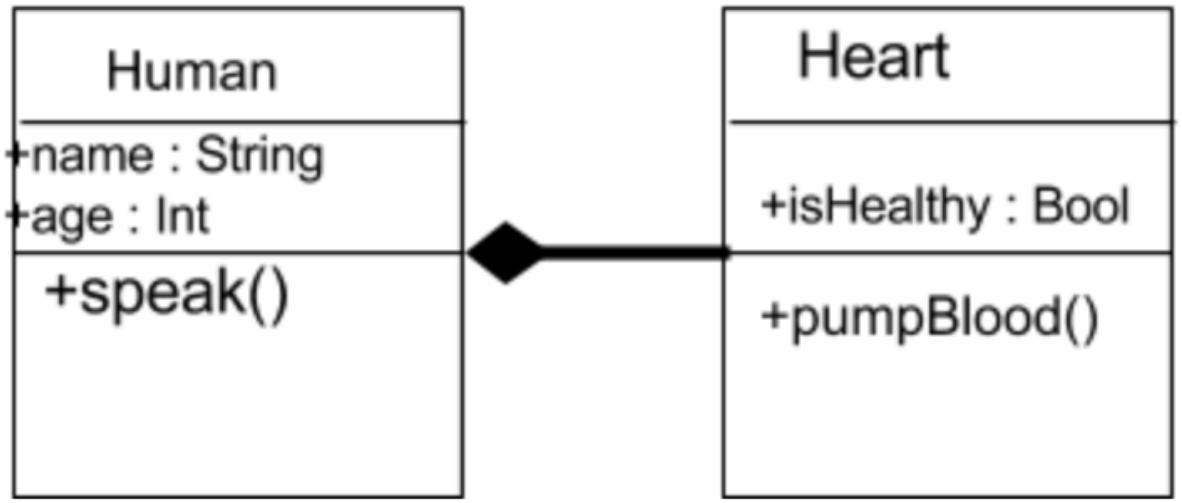
Realization/Implementation

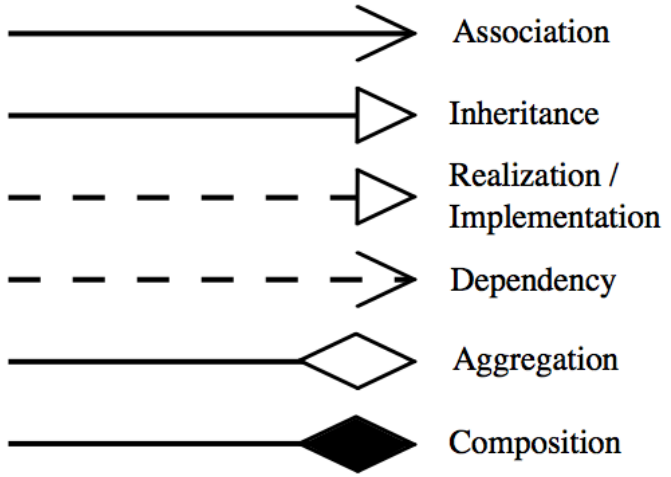


Aggregation

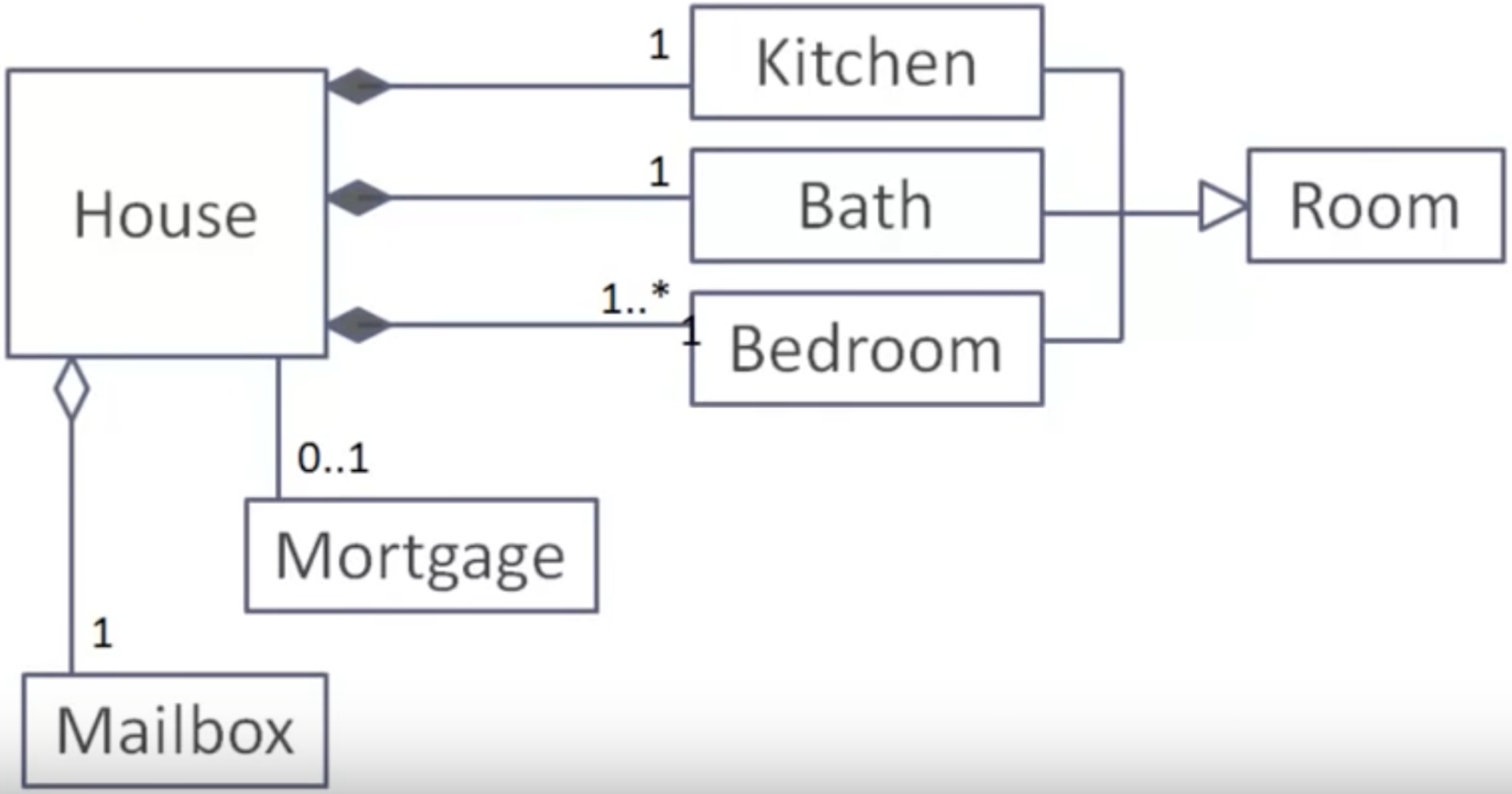
- “has a”
- “is part of”

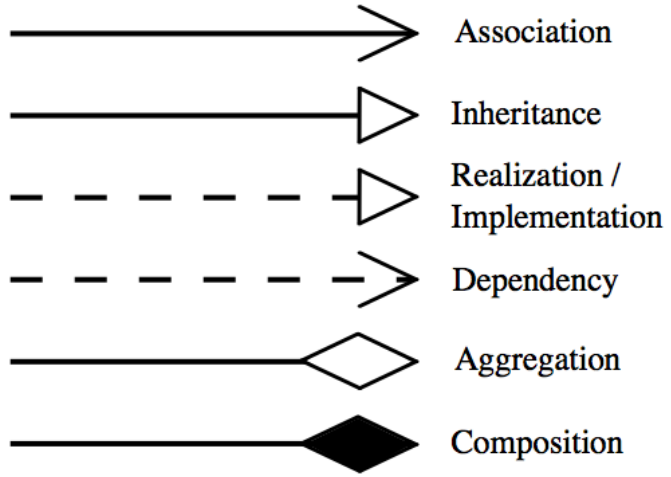
Composition





Multiplicity





OO modeling points

- Class name should be nouns
- Verbs become operations

Analysis vs Design

- Class diagrams are used in both analysis and design
- Analysis - conceptual
  - model problem, not software solution
  - can include actors outside system
- Design - specification
  - tells how the system should act
- Design – implementation
  - actual classes of implementation

Class Diagram

Central model for OO systems

- Describes data and behavior
- In UML, is used along with Use Cases and

Packages for analysis

- Is also used to describe implementation
- Don’t confuse analysis and implementation!

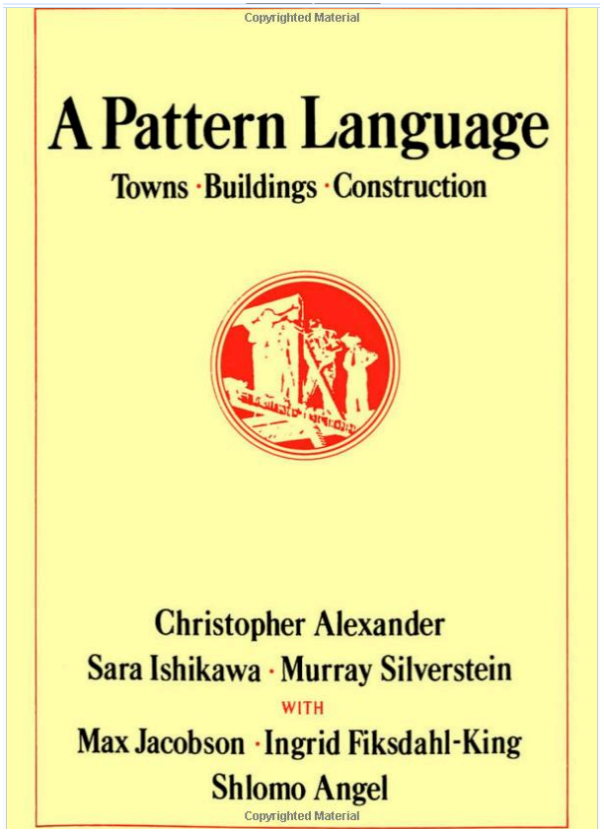
Don’t add all implementation details

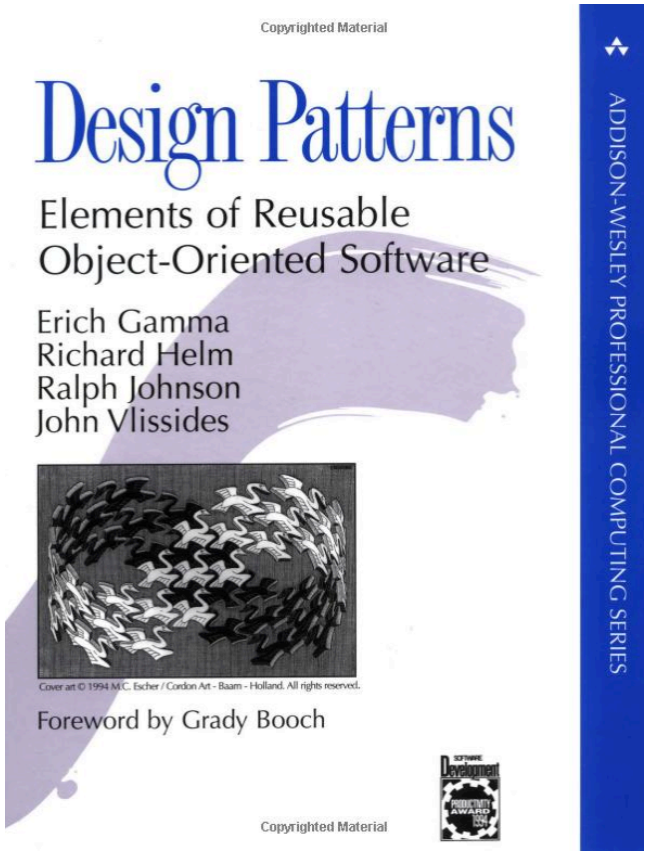
Readings

- More UML resources
- http://dn.codegear.com/article/31863
- http://www.sparxsystems.com.au/
- UML_Tutorial.htm

  http://www.gnome.org/projects/dia/umltut/index.html

History of Patterns





- Elements of Reusable Object-Oriented Software
- 23 OO patterns

Design Patterns

- When used strategically, they can make a programmer significantly more efficient by allowing them to avoid reinventing the proverbial wheel, instead using methods refined by others already
- Provide a useful common language to conceptualize repeated problems and solutions when discussing with others or managing code in larger teams.

Classification of patterns

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

Criticism of Design Patterns

- **Kludges for a weak programming language**

Usually the need for patterns arises when people choose a programming language or a technology that lacks the necessary level of abstraction.

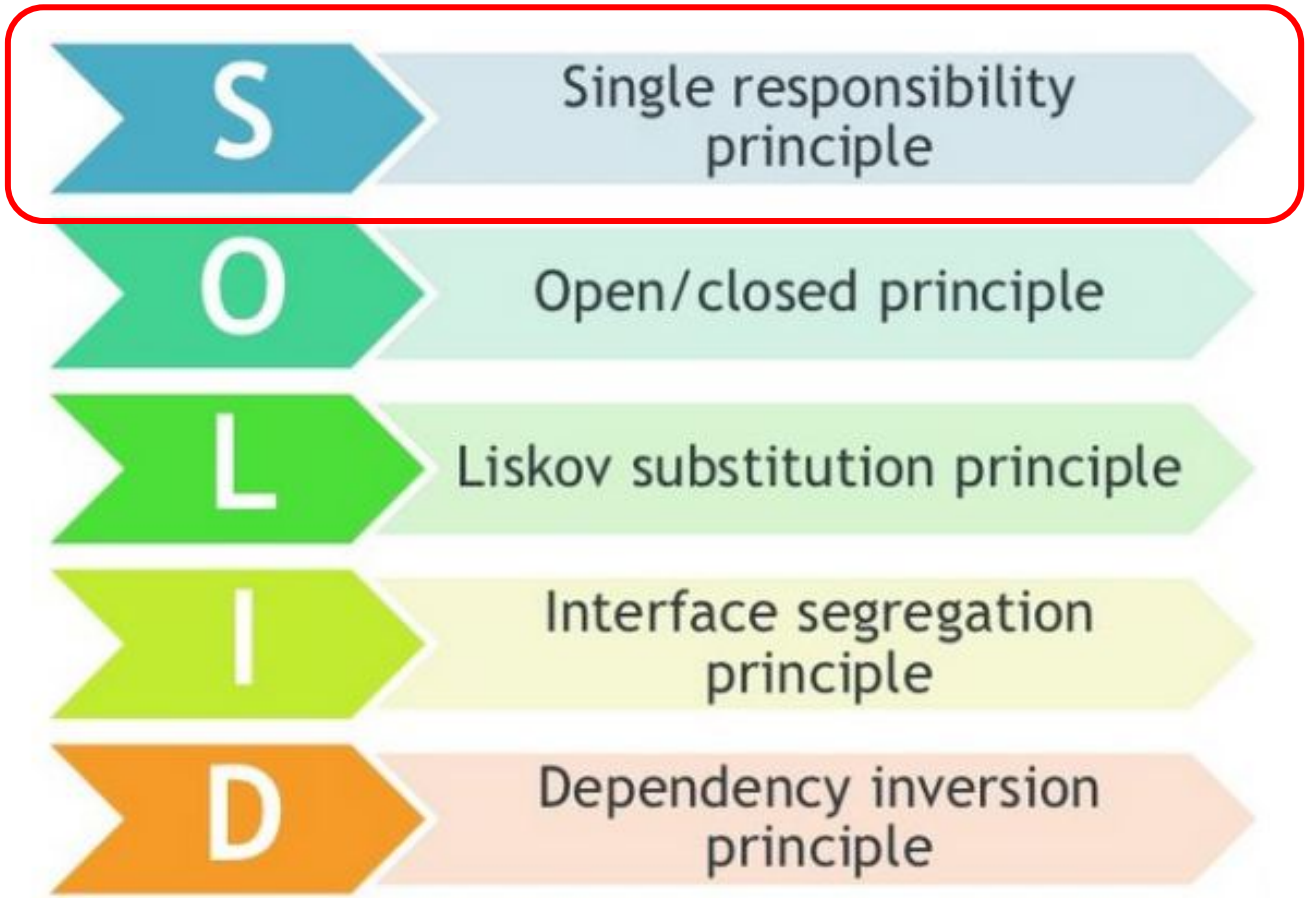
- **Inefficient solutions**

Patterns try to systematize approaches that are already widely used.

- **Unjustified use**

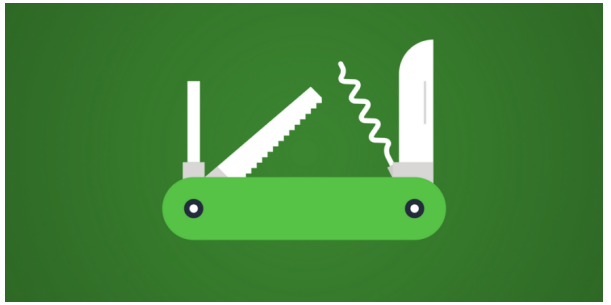
If all you have is a hammer, everything looks like a nail.

OO Design Principles

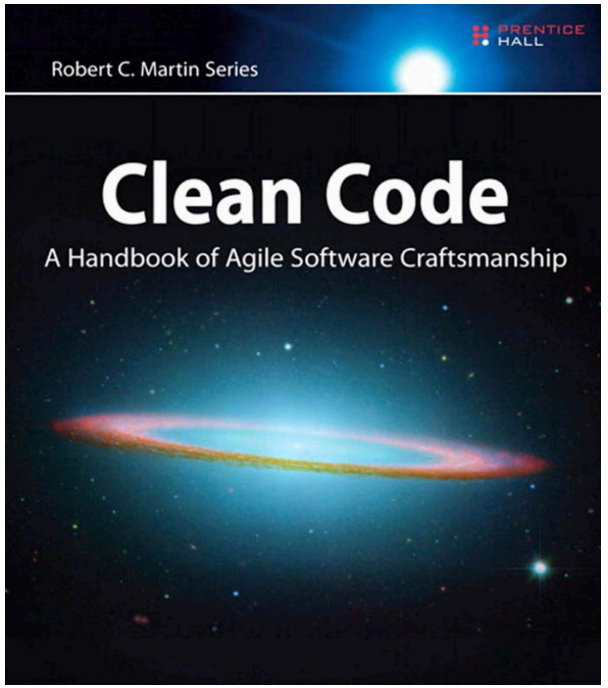


OO Design Principles

Single Responsibility Principle



A class should have one, and only one, reason to change.
Just because you can, doesn't mean you should

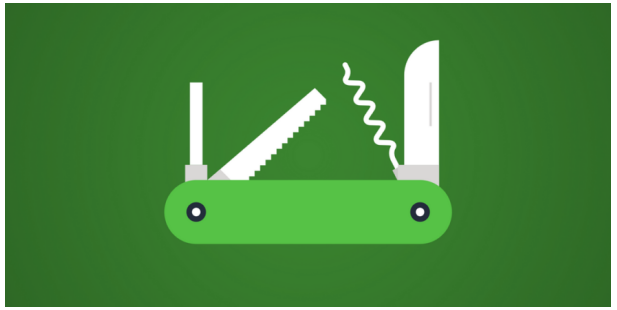


Benefits:

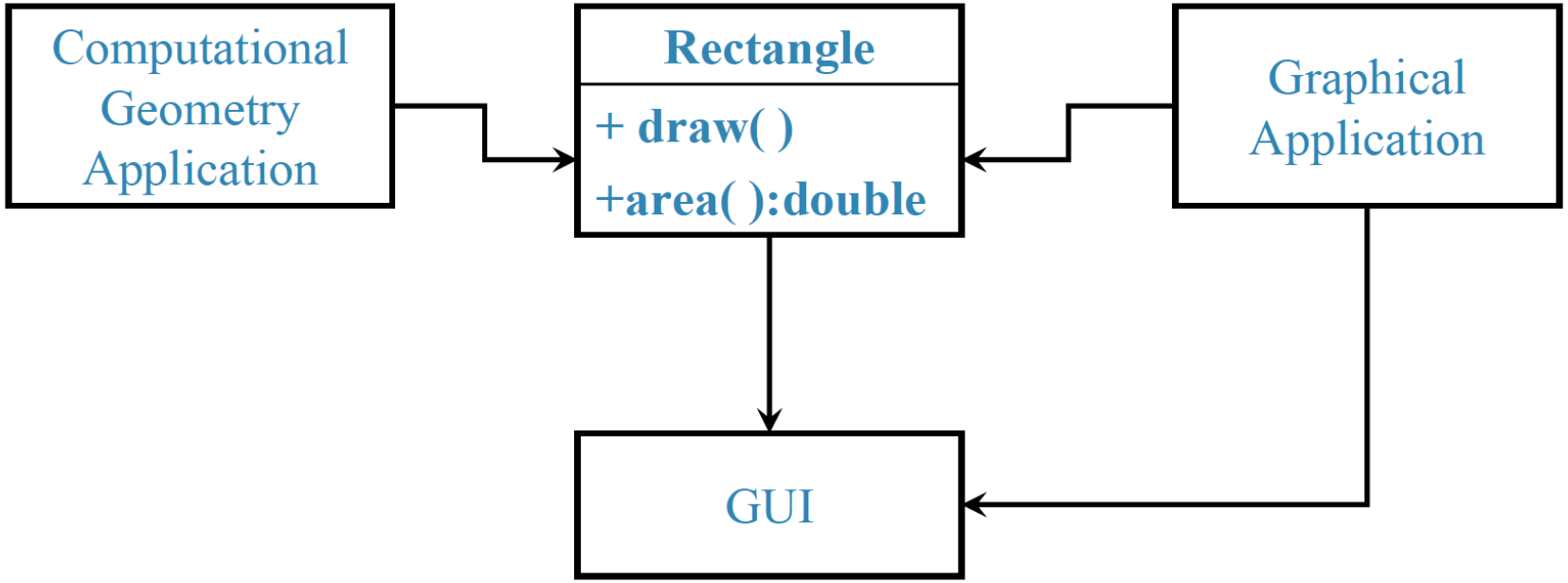
- Frequency and Effects of Changes
- Easier to Understand

Q: What is the responsibility of your class/component/microservice?

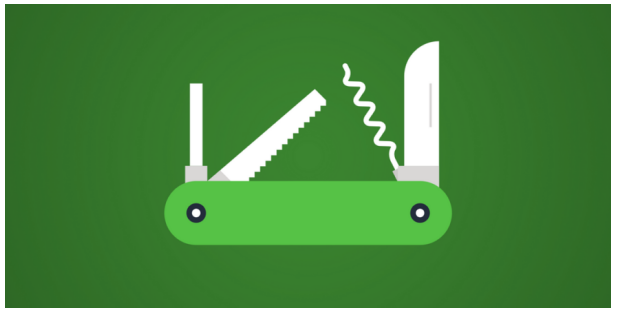
Single Responsibility Principle

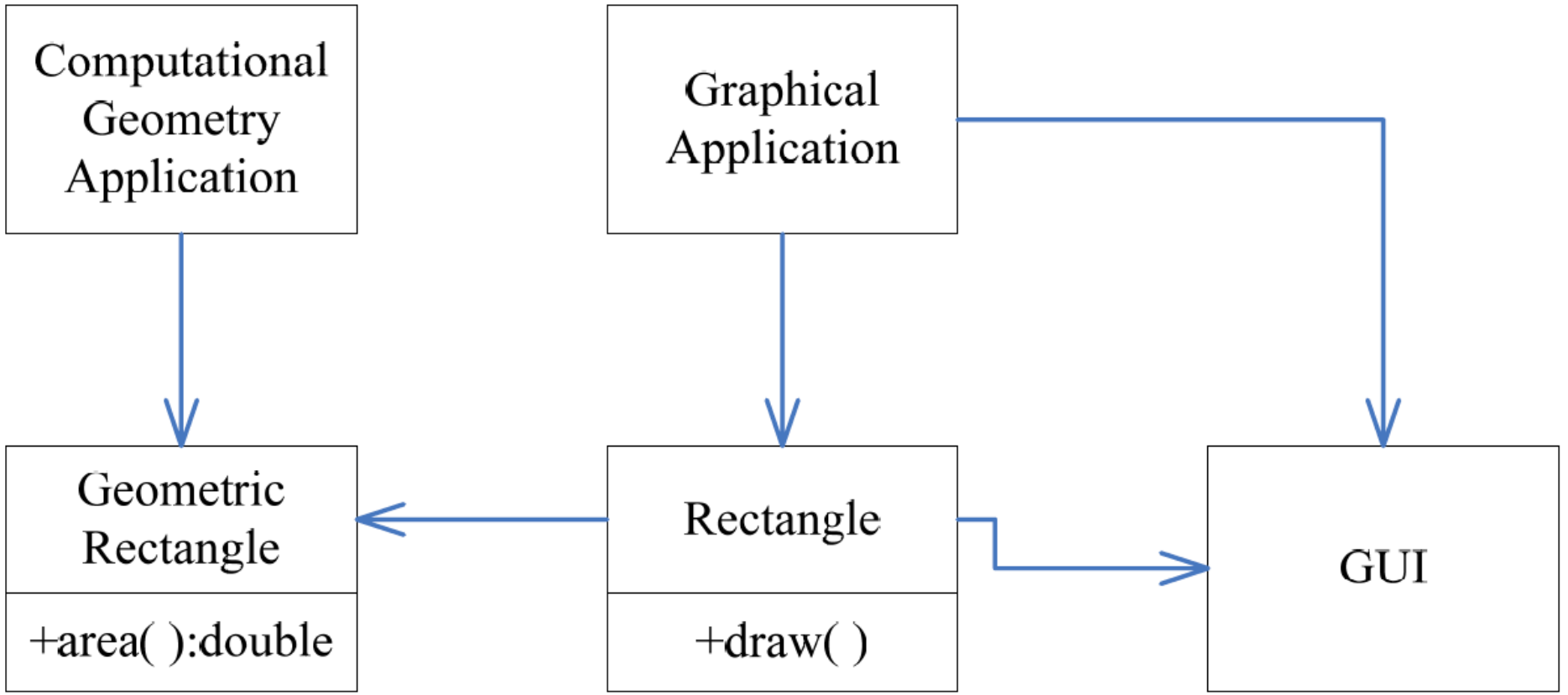


A class should have one, and only one, reason to change.



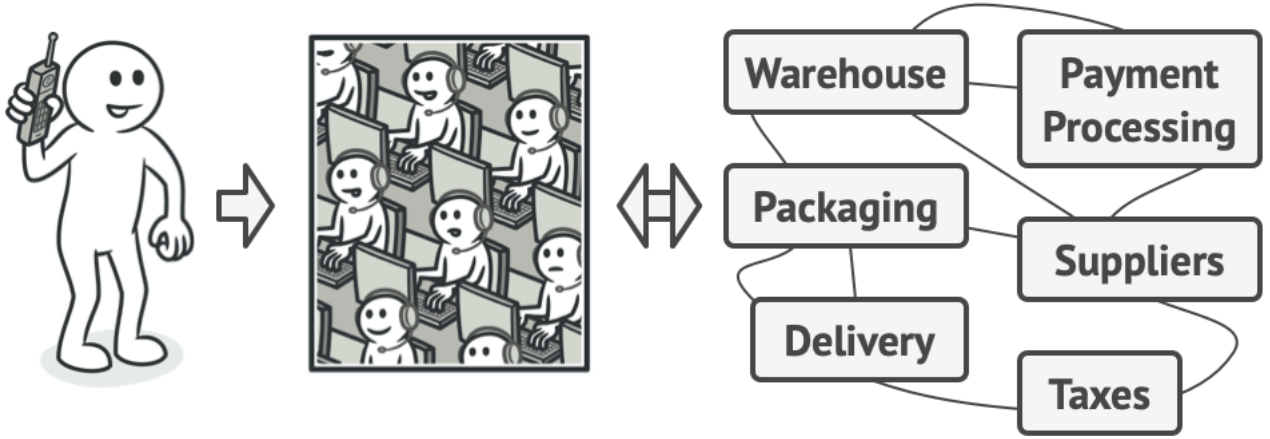
Single Responsibility Principle



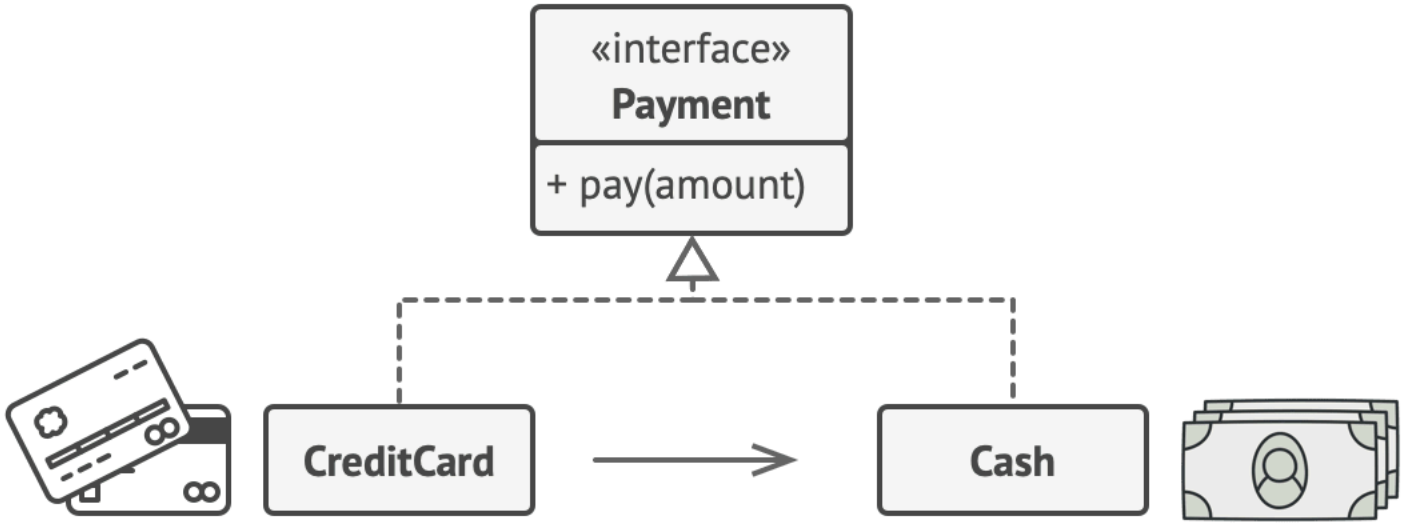


Corresponding Design Patterns

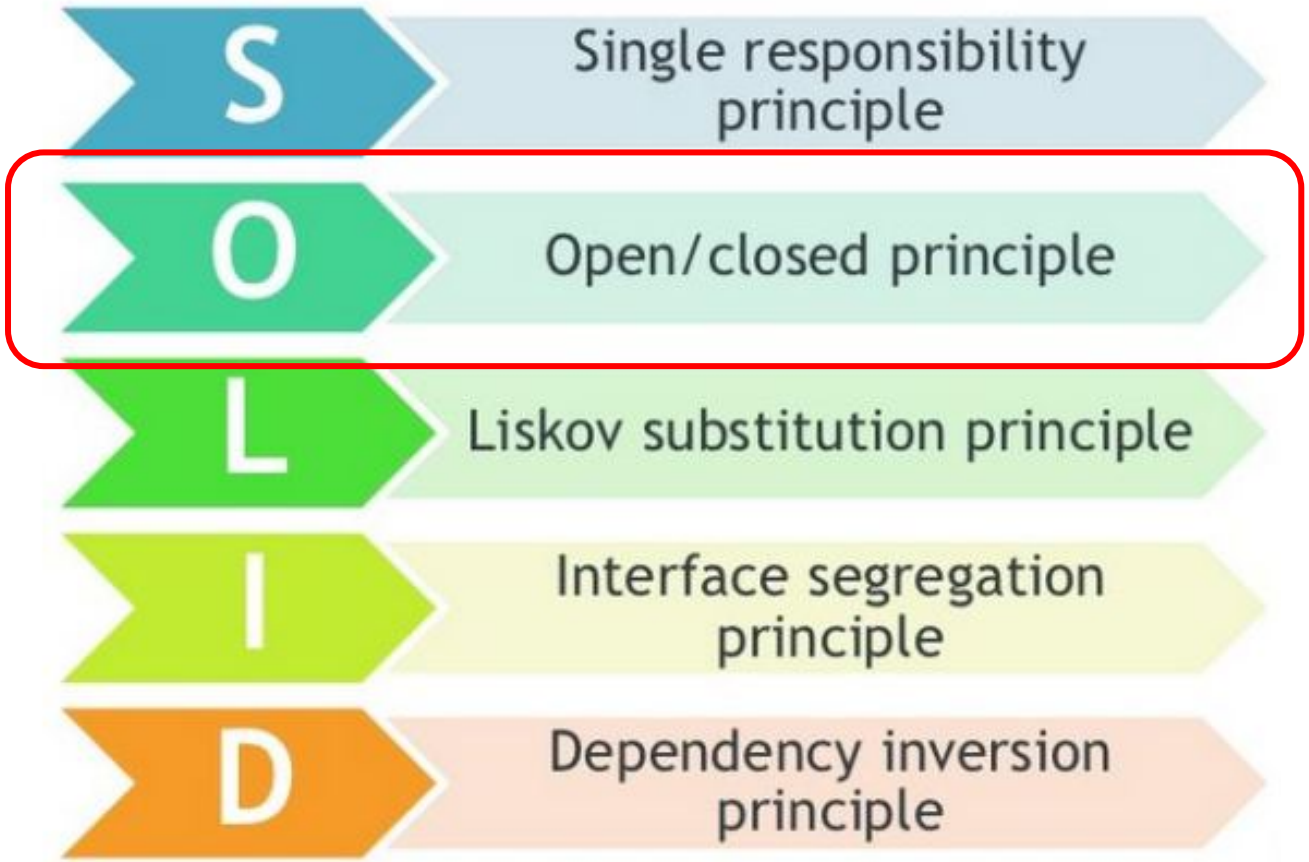
- Façade



- Proxy



OO Design Principles



Open-Closed Principle (OCP)

- Software entities should be open for extension, but closed for modification.

Open-Closed Principle

- Implementation:
  - inheritance
  - composition
- Benefits:
  - extend a component's logic without breaking backward compatibility
  - test different component implementations (that have the same logic) against each other.

Thoughts? Critiques on OCP

- Adding un-needed flexibility to code (to make it open for extension) breeds complexity and carrying cost.
- It requires imagining all sorts of use-cases that don’t exist in order to make it ultimately flexible.
- Principle != you should always do this

Corresponding Design Patterns

- Strategy
- Simple Factory
- Factory Method
- Abstract Factory
- Builder
- Bridge
- Façade
- Mediator