System Design & Analysis

LECTURE 14

Object Oriented Programming

- The system is organized as a set of classes & objects
- Classes & objects are associated in different ways
- The implementation is done in an object oriented programming languages
- OO languages are build upon the elements if object model.

Elements of objects Model

- OO languages build upon these elements support object oriented programming
  - Abstraction
  - Encapsulation
  - Hierarchy
  - Polymorphism

Abstraction

- Donates essential characteristics of an object
  - Distinguish it from other objects
  - Relative to the perspective of the viewer
- It a simplified view or specification that emphasizes some details while suppressing/ignoring other
- Unnecessary details are left out
  - They may not be relevant to the problem
- A class/struct/interface is an abstraction

Example

SCHOOL

Name
RollNo
Class
Registration No.
Marks
GetGrade
CalculateTotalScore

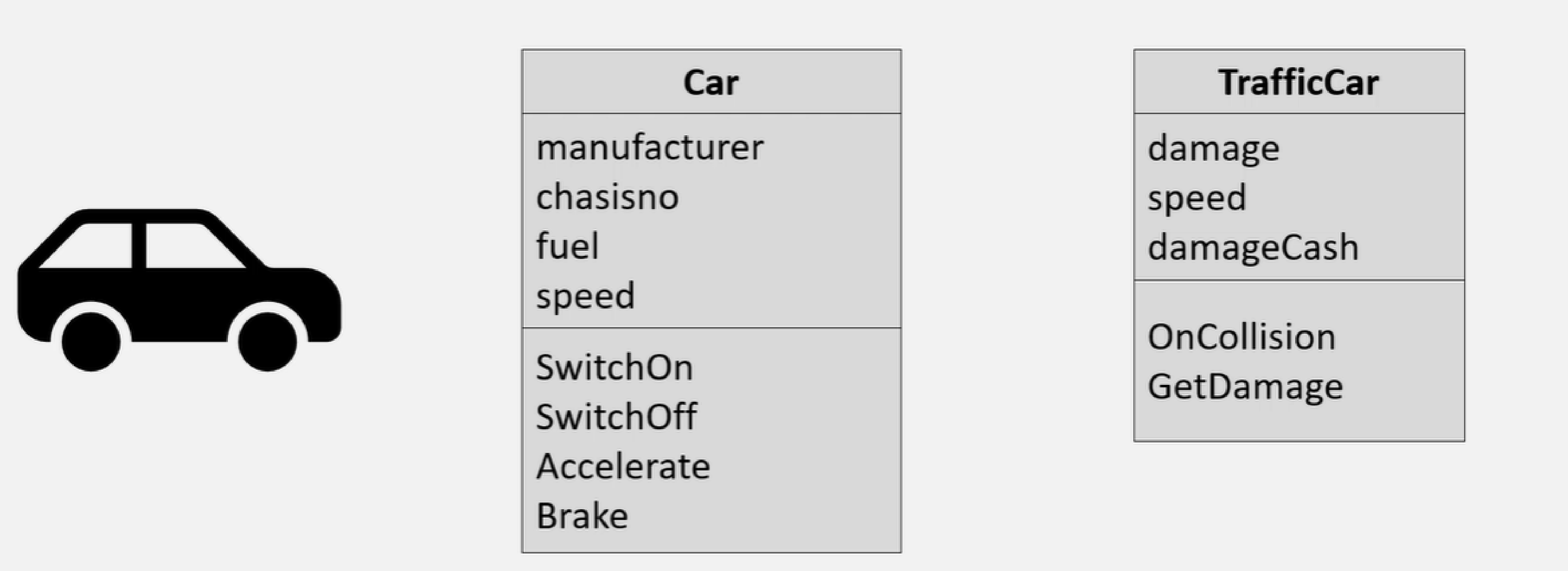
INNOCULATION

Name
Age
Weight
Gender
Allergies
School
History
AddRecord
GetRecord

Abstraction

- Abstraction is performed on the domain
- Performing correct abstraction for a given domain is necessary
- Domain is expertise is important for performing correct abstraction
- Focus on entity abstraction
  - Represents a useful model of domain entity
  - Closely matches with the vocabulary of the domain
- e.g bank, Account, transaction, etc

Example



Car

manufacturer
chasisno
fuel
speed

SwitchOn
SwitchOff
Accelerate
Brake

TrafficCar

damage
speed
damageCash

OnCollision
GetDamage

Object Responsibility

- The abstraction provides the services or uses service of some other object
  - This forms the behavior of the abstraction
  - Provides an outside view
- This view defines a contract that others depend on
- This behavior forms the responsibility of the abstraction
  - Provided as operations, methods or member function
  - Collectively called protocol
  - Protocol forms the static and dynamic view of the abstraction
  - Included static & dynamic of abstraction

Advantages

- Resolves complexity
- Makes it simpler to model a solution for a problem
- Unnecessary details are left out
- Focuses on the details relevant to the problem domain
- Represents real-life objects in software as domain entities

Encapsulation

- The elements of abstraction provide a behavior of the object through implementation
- Upholds the contract of its behaviour
- This implementation should be treated as a secret
  - Kept hidden from clients
- This is achieved through encapsulation

Encapsulation

- Hides the implementation details of an object
- Focuses on the implementation that gives rise to the behavior
- Achieved through information hiding
- Structure and implementation of the methods is hidden
- Client only knows about the contract
- Shielded from implementation details at lower level of abstraction

Encapsulation

- Abstraction works only when encapsulated
- Every class will have two parts: an interface and an implementation
- Interface captures the outside view and provides the behavior
- Clients will make assumptions based on the outside view
- Implementation is encapsulated, so clients cannot make any assumptions
- In turn, the abstraction is required to be accountable for its behaviour

Advantages

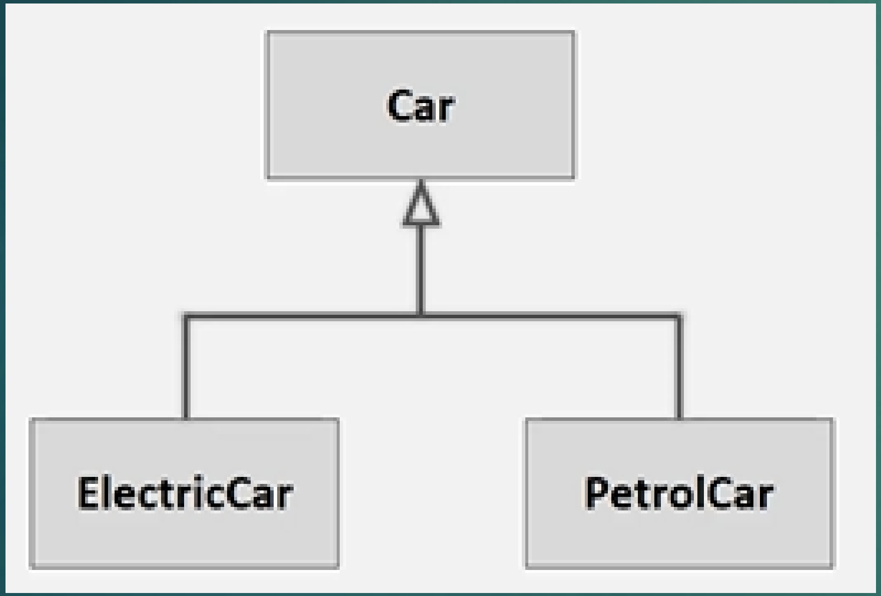
- Clients shouldn’t need to see the implementation details
- Change in internal implementation doesn’t effect the clients
- Encapsulation decouples clients from implementation details
- Enforces state changes to object only through behavior
- Helps in building scalable & flexible system

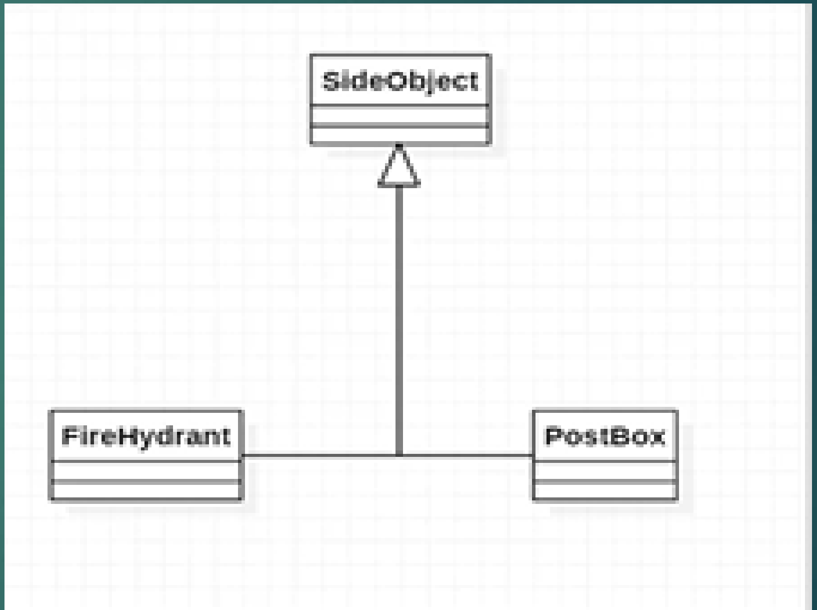
Hierarchy

- Abstraction helps represent domain objects to resolve complexity
- Encapsulation further hides the implementation details
- The complexity can be reduced further by creating hierarchy of abstractions
  - Hierarchy represents relationship between abstractions
- Helps represents the problem domain and its objects
- The important hierarchies in a complex system are
  - Inheritance
  - composition

Inheritance

- Is-a relationship or generalization
- Represents a relationship where one class is a kind of other
- A class (base or super class) will share its structure and behavior with another class (child class or subclass)
- The base class contains important behavior which must be exhibited by other classes
- Any class can inherit the behavior from the base class
- May provide a specialized implementation of the behavior
- Inheritance implies generalization/specialization hierarchy


Car
ElectricCar
PetrolCar

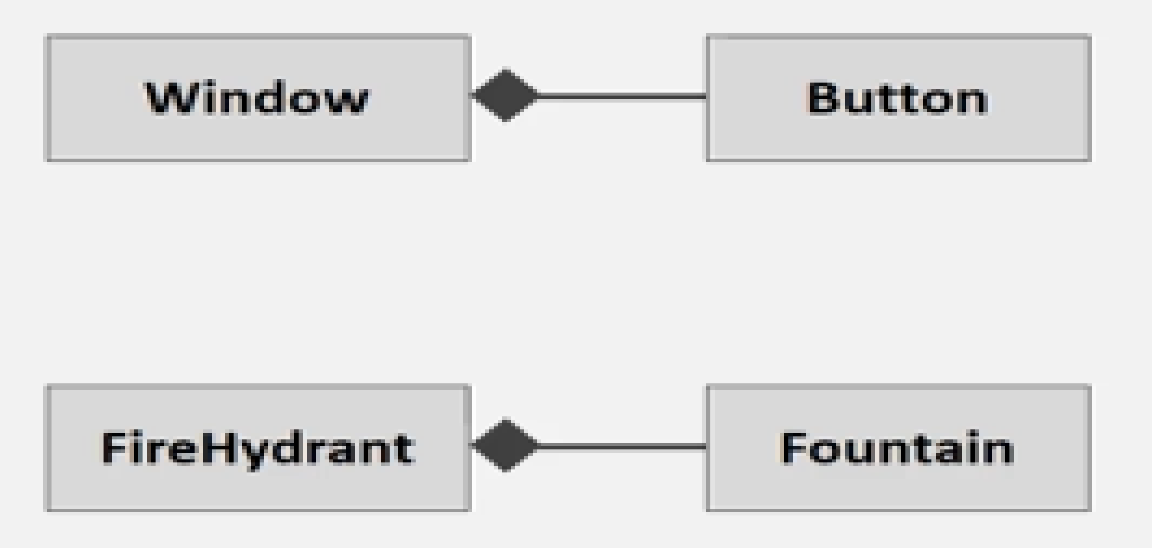

SideObject
FireHydrant
PostBox

Containment

- This is relationship between objects
- Signifies “has a” relationship
- One object may contain another object to reuse its behavior
- Multiple forms of containment exists
  - Composition
  - Aggregation
  - Association
  - Dependency

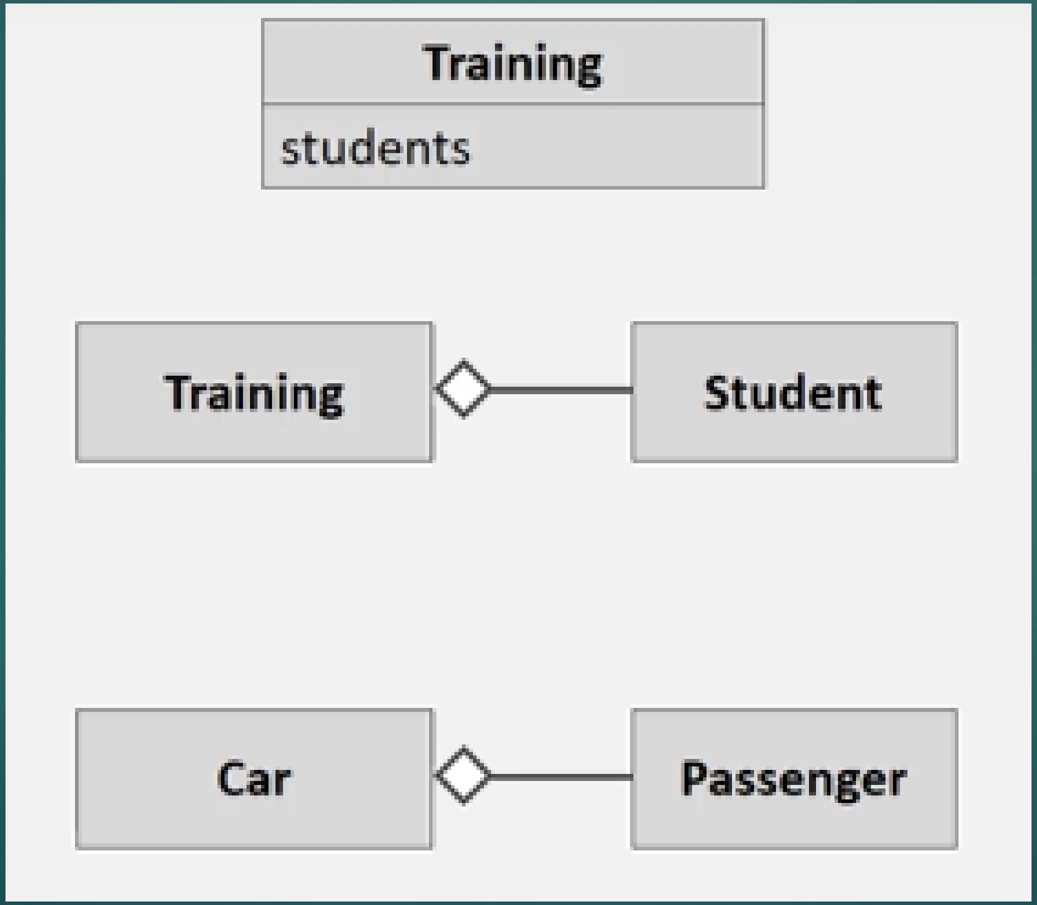
Composition

- This is a strong relationship & signifies a physical containment
- One object is part of another object
- The outer object is responsible for its lifetime
- E.g. Button is a part of window



Aggregation

- Donates a logical containment of an object
- Weaker than composition
  - E.g. Training has students
- Training does not physically contain the students
- The students are shared with other trainings
- A training does not control the lifetime of its students; they exist independently



Training

students

Training

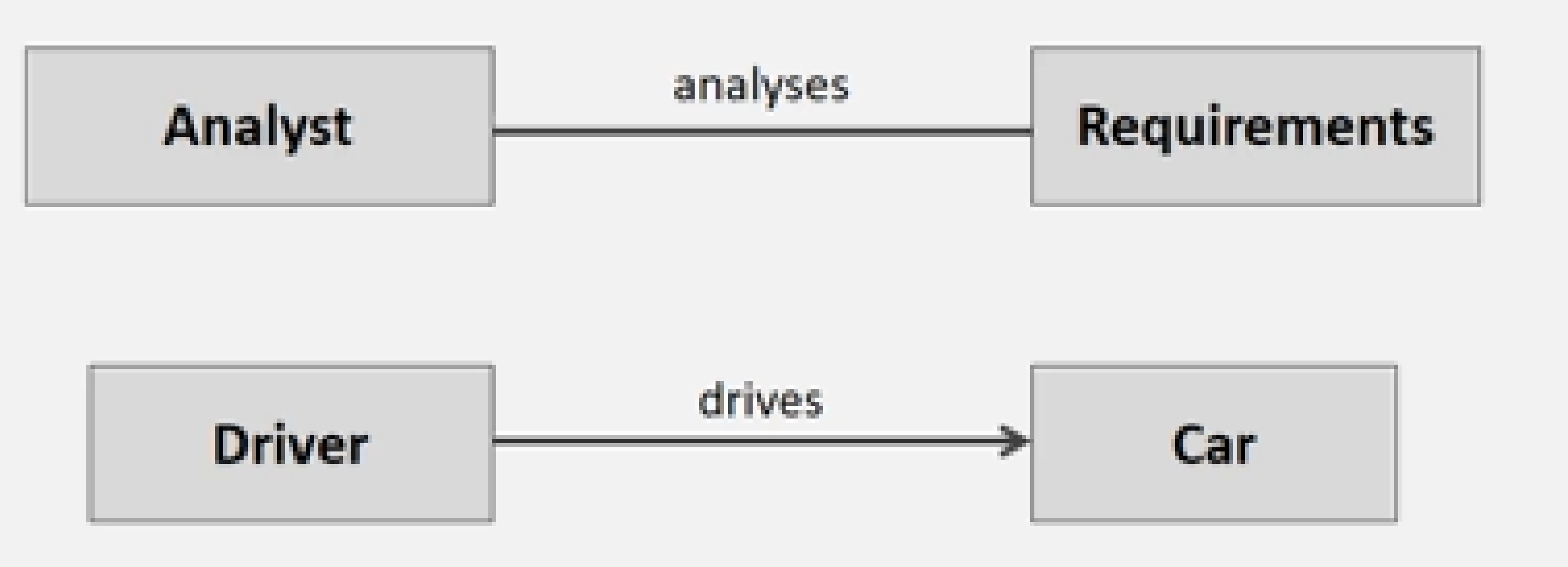
Student

Car

Passenger

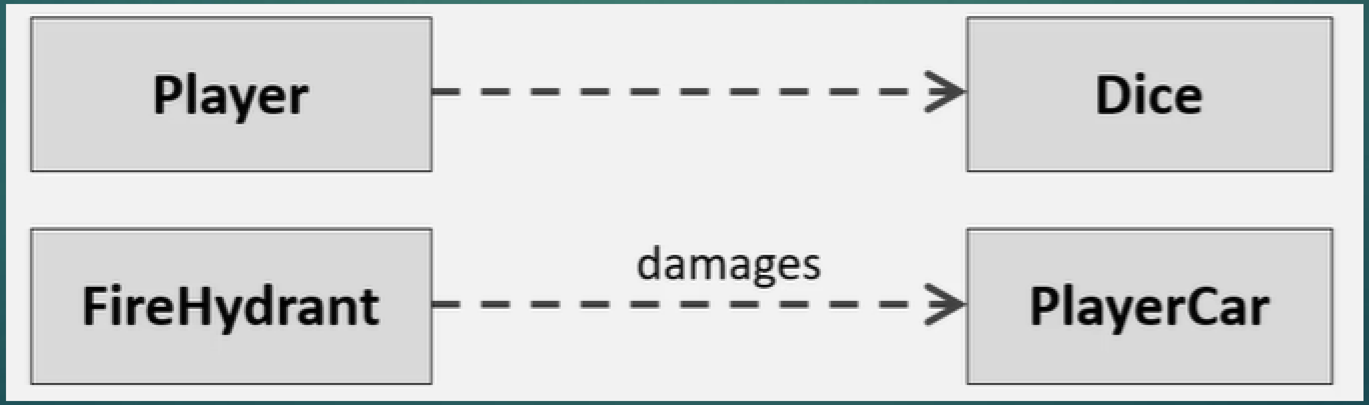
Association

- This also implies reuse of objects
- Represents a semantic connection between the class
- A class will contain a reference of another object and use it for some time
- Often accompanied with phrase such as uses, controls, etc.
- E.g. Analyst analyses requirements

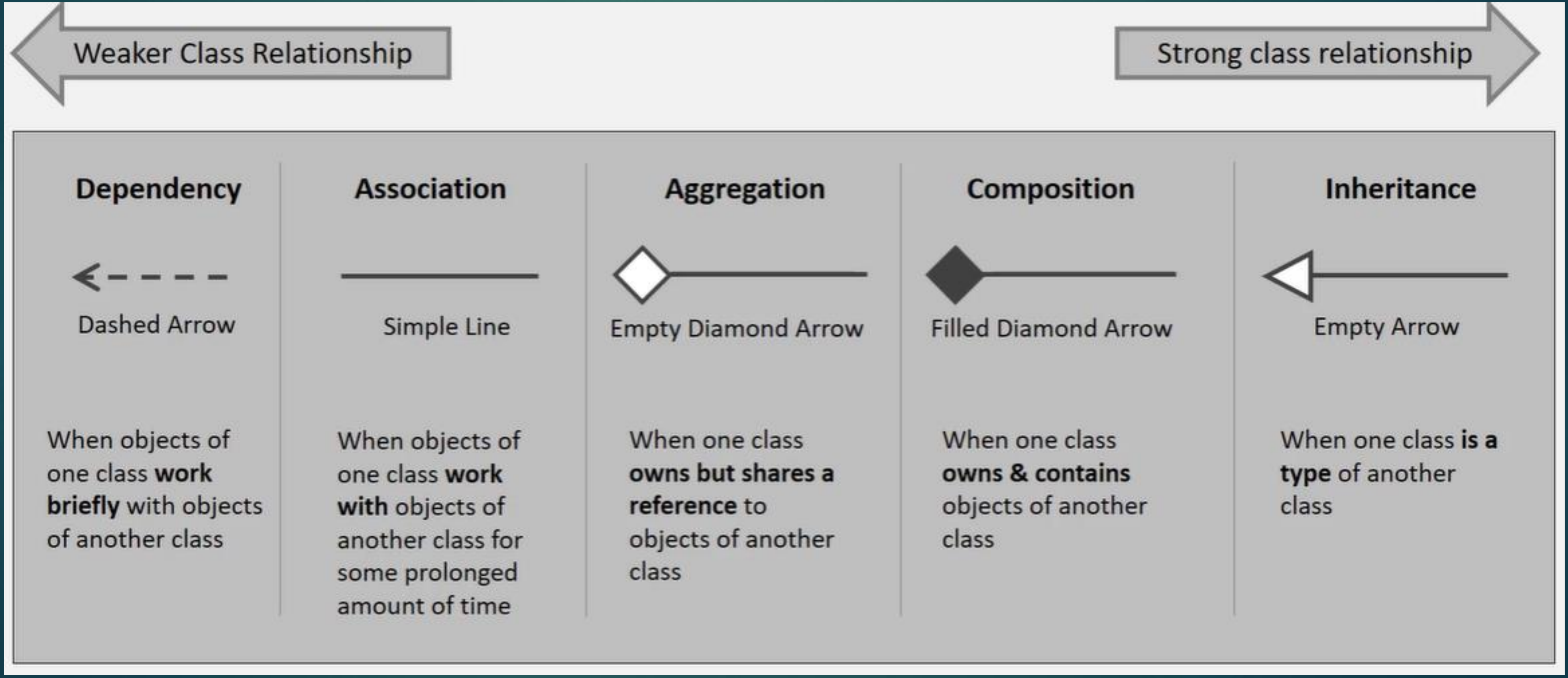

Analyst
analyses
Requirements
Driver
drives
Car

Dependency

- Weakest form of relations
- This relationship is formed when an object works with another object briefly
- A class method accepts an object of another class
- E,g, a player rolls a Dice



Relationship Type



Weaker Class Relationship

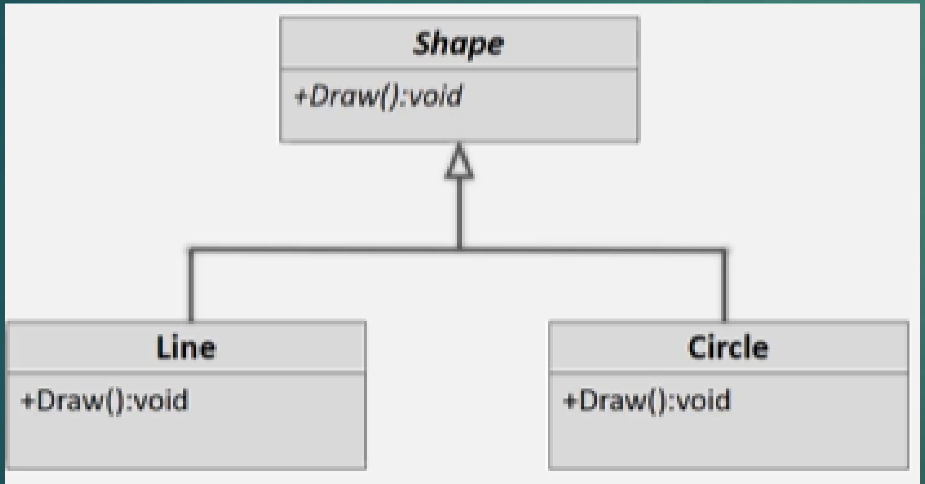
Strong class relationship

Dependency	Association	Aggregation	Composition	Inheritance
Dashed Arrow	Simple Line	Empty Diamond Arrow	Filled Diamond Arrow	Empty Arrow
When objects of one class **work briefly** with objects of another class	When objects of one class **work with** objects of another class for some prolonged amount of time	When one class **owns but shares a reference** to objects of another class	When one class **owns & contains** objects of another class	When one class **is a type** of another class

Polymorphism

- Means different forms
- These forms represent different implementation of the same behavior in different objects
- The are united via inheritance
- The behavior is invoked on the base/super object, but the action is performed on its child object
- The child class overrides the implementation of the base class

- OO languages support polymorphism in many ways
- Compile time polymorphism
- Runtime polymorphism



Shape

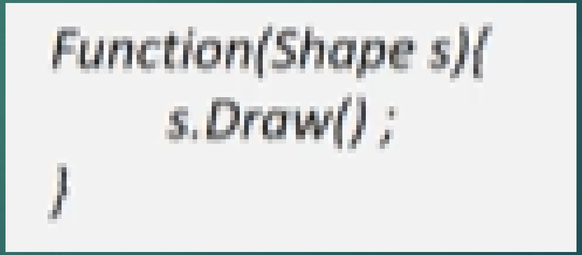
+Draw():void

Line

+Draw():void

Circle

+Draw():void



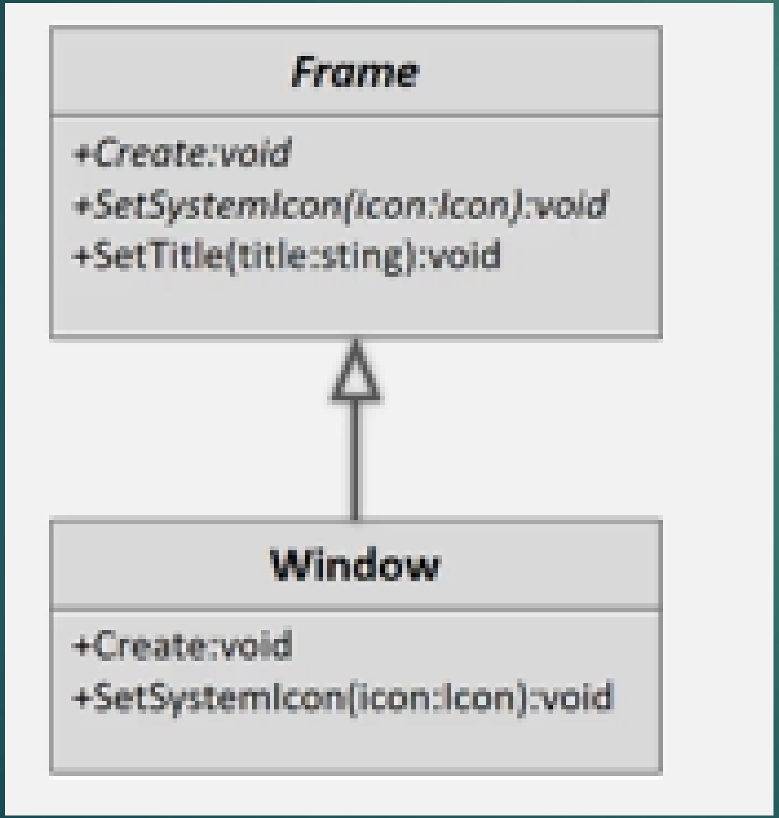
```
Function(Shape s)(
        s.Draw() ;
}
```

Advantages

- Prompt reuse
- More classes can be added without requiring change to existing code
- Certain implementation may not even require a recompilation of the binary
- Works in tandem with containment and inheritance

Abstract Class

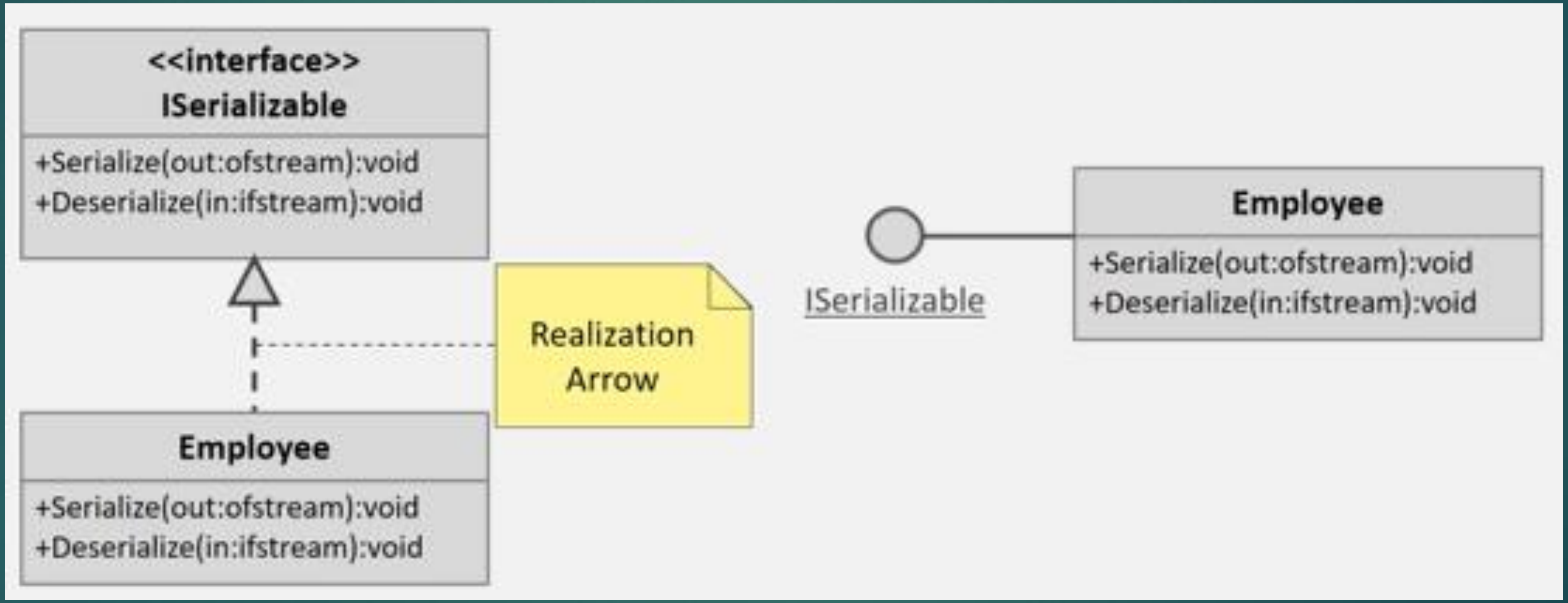
- Only have method name not the implementation
- These methods exist to provide only the behavior that the subclass must implement
- Such methods are abstract (or pure virtual function in C++) and the class also becomes an abstract class
- An abstract class cannot be instantiated
- It can have non-abstract methods, fields, static members, etc.
- The subclass will become abstract if it does not override abstract methods from base class


Frame
+*Create:void*
+*SetSystemIcon(icon:Icon):void*
+SetTitle(title:sting):void
Window
+Create:void
+SetSystemIcon(icon:Icon):void

Abstract class name and methods are written in abstract

Interface

- Collection of operations that do not have any implementation
- Similar to abstract class that has all methods as abstract (or pure virtual)
- Cannot be instantiated
- All methods are implicitly public
- The methods have to be implemented by the child classes
- Interfaces are used as connections between application or libraries or components


<<interface>>
ISerializable
+Serialize(out:ofstream):void
+Deserialize(in:ifstream):void
Realization Arrow
Employee
+Serialize(out:ofstream):void
+Deserialize(in:ifstream):void
ISerializable
Employee
+Serialize(out:ofstream):void
+Deserialize(in:ifstream):void

Abstract class Vs. Interface

Abstract Class

- Can contain fields or methods
- May provide default implementation of some behavior
- Subclasses extend the behavior of the base
- Subclasses cannot inherit from multiple abstract classes
- Can have different access modifiers
- Use to represent common behavior and implementation when different subclasses are related

Interface

- Contains only behavior
- Cannot contain any implementation
- Subclasses implement the behavior of the interface
- subclasses can implement multiple interfaces
- All methods are public
- Use to represent common behavior that “must” be implemented by disparate classes