Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Overview of OO Analysis and Design

Google Cloud Platform

Windows Azure

amazon web services

APACHE kafka
A distributed streaming platform

Apache Software Foundation

APACHE Spark

Java

hadoop

Overview of OO Analysis and Design.

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Module1: Inheritance, Association, Delegation.

Module2: OOAD's SOLID principles

Module3: Types of classes

Module4: Design patterns

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Module 1

Inheritance, Association, Delegation.

Some Characteristics of OO Systems

- Externally visible behavior of an object
- The internal data structure and implementation that gives rise to the externally visible behavior of an object
- Set of public functions of a class

Abstraction, Encapsulation, and Interface

Abstraction = what an object is / what an object does

Example: ATM device

- Dispenses cash to a valid customer
- Reports current balance of a customer
- Allows funds transfer from one account to another

Encapsulation = how an object does what it does

Example: ATM device

- How it connects to the bank's database to verify balance
- How it counts currency to dispense

Interface = how the clients of an object talk to it

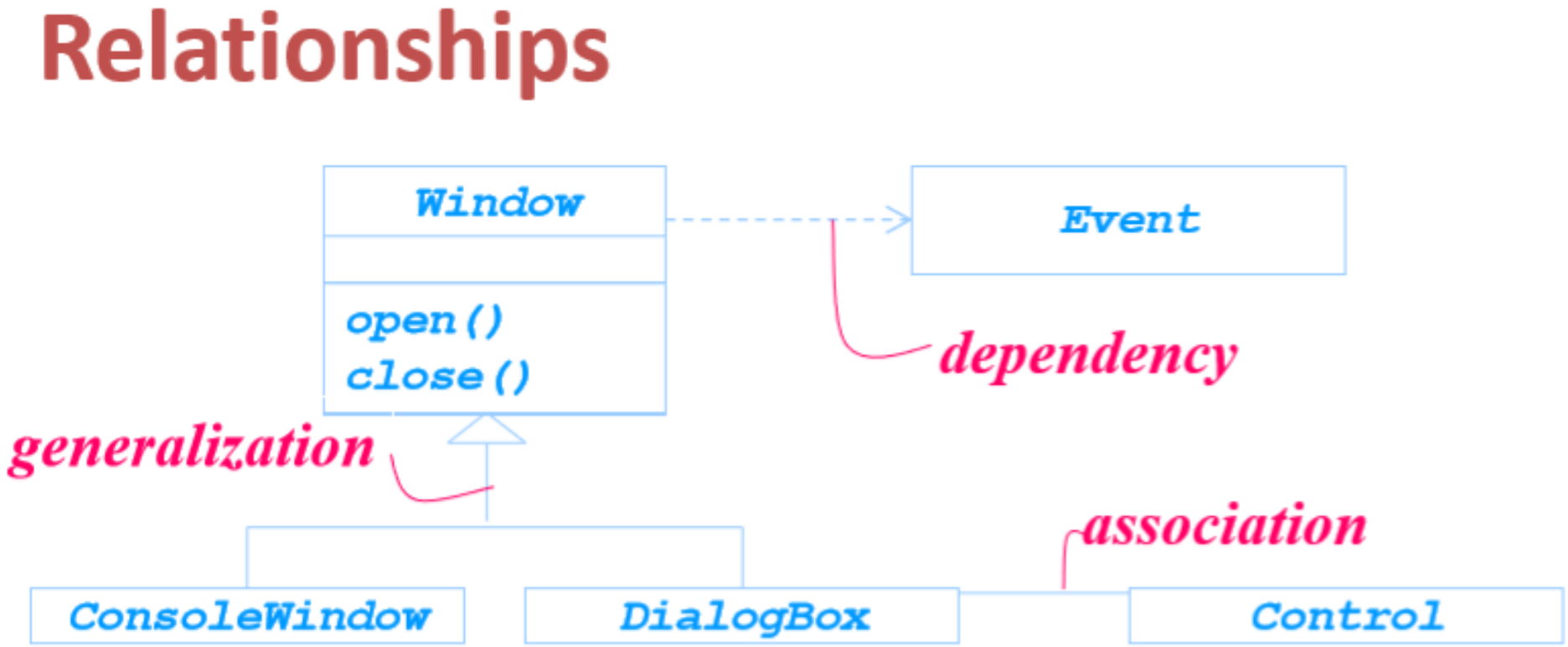
Example: ATM device

- The slot for inserting the ATM card
- The buttons for the users to specify their options

Relationship

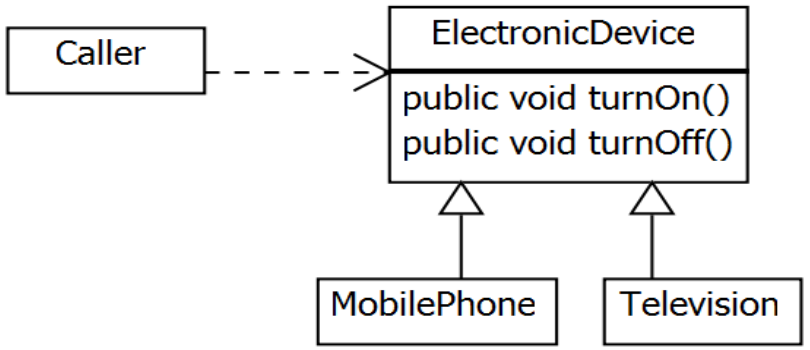
Association is a relationship between two objects.
Aggregation is a special form of association. When an object **‘has-a’** another object,
Composition is a special form of aggregation. When an object contains the other object, if the contained object cannot exist without the existence of container object.

Relationship -



Modeling Business with Classes

- **Inheritance** (or subclassing) is an essential feature of the Java programming language. Inheritance provides code reuse through:
  - **Method inheritance**: Subclasses avoid code duplication by inheriting method implementations.
  - **Generalization**: Code that is designed to rely on the most generic type possible is easier to maintain.



Class Inheritance Diagram

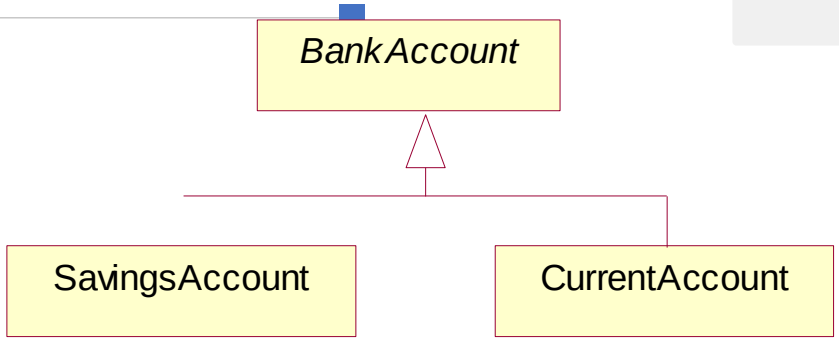
Relationship – Inheritance , Composition and Aggregation

- Complex class structures can be built through inheritance, composition, and aggregation.
- **Inheritance** establishes an "**is-a"** relationship between classes.
  - The subclass has the same features and functionality as its base class, with some extensions.
  - A Car **is-a** Vehicle. An Apple **is-a** Food. A Customer **is-a** Person
- **Composition and aggregation** are the construction of complex classes that incorporate other objects.
  - They establish a "has-a" relationship between classes.
  - A Car **has-a** Engine. A Customer **has-a** CreditCard.

Relationship – Inheritance , Composition and Aggregation

- **In *composition***, the component object **exists solely** for the use of its composite object.
  - If the composite object is destroyed, the component object is destroyed as well.
  - For example, an Employee **has-a** name, implemented as a String object. There is no need to retain the String object once the Employee object has been destroyed.
- **In *aggregation***, the component object can (but isn't required to) have **an existence independent** of its use in the aggregate object.
  - Depending on the structure of the aggregate, destroying the aggregate may or may not destroy the component.
  - A Customer **has-a** BankAccount object. However, the BankAccount might represent a joint account owned by multiple Customers. In that case, it might not be appropriate to delete the BankAccount object just because one Customer object is deleted from the system

Inheritance

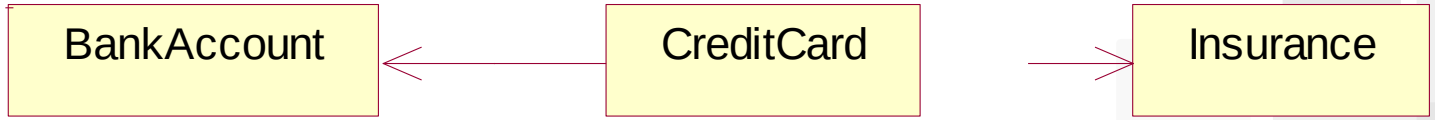



```
class SomeClass {
   public void someFunction (BankAccount acc) {…}
}
```

Client code:

```
SomeClass someObject = new SomeClass ();
someObject.someFunction (new SavingsAccount ());
```

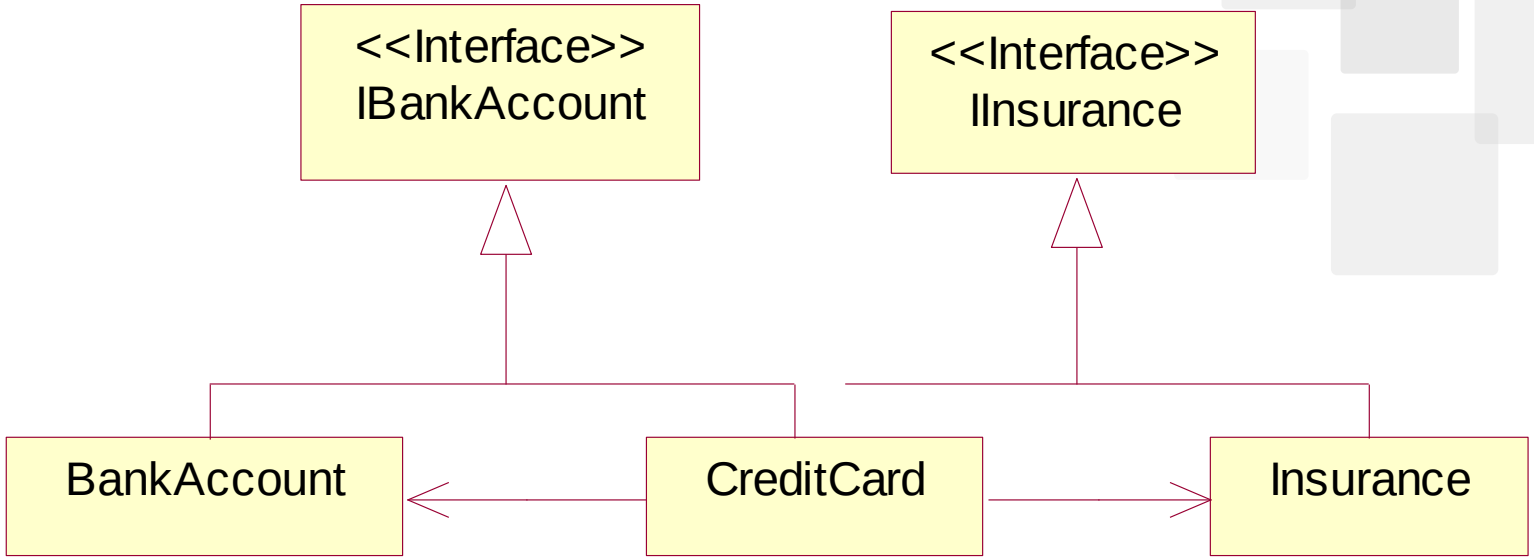
Delegation



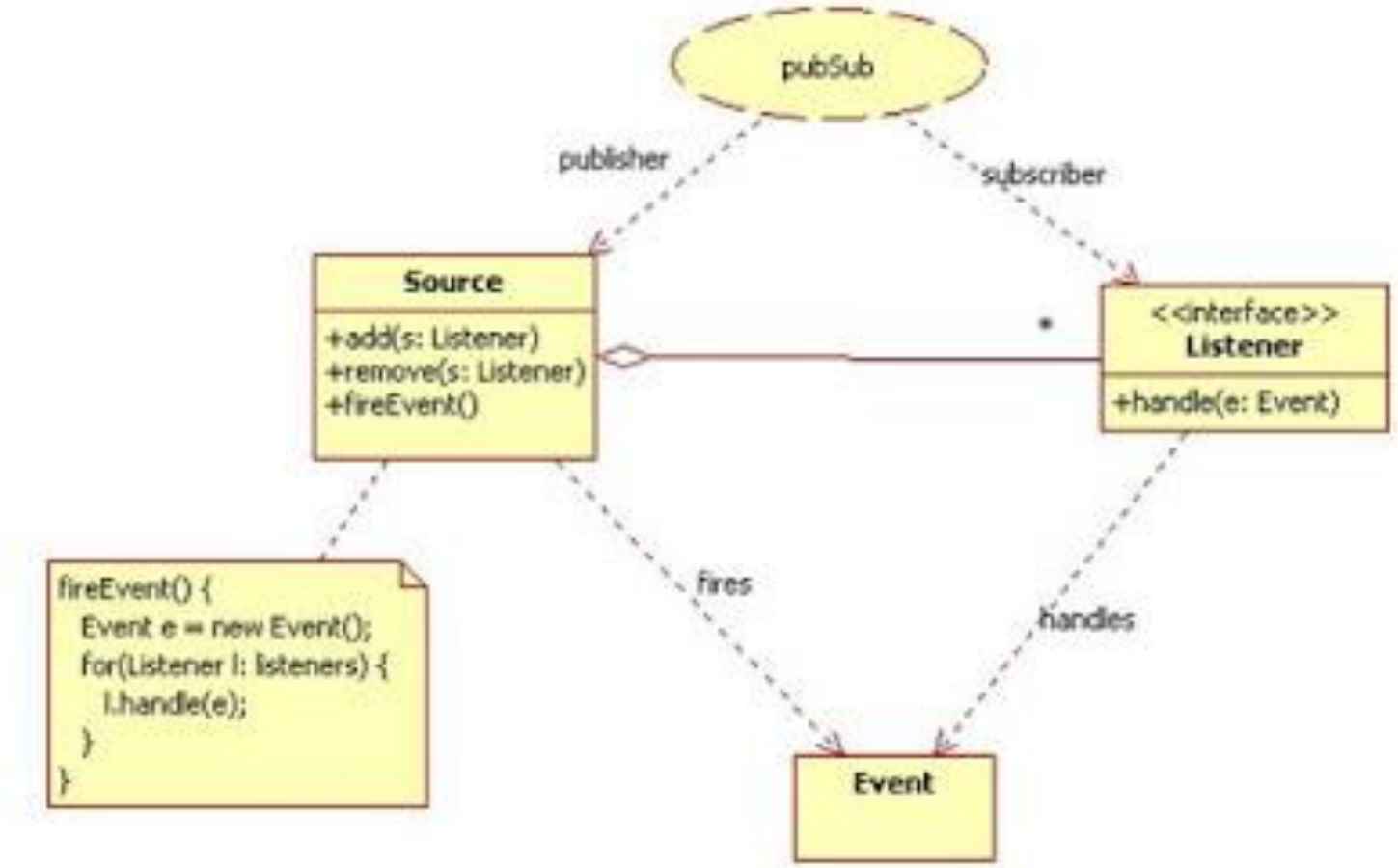

```
class CreditCard {
  private BankAccount bankAccount;
  private Insurance insurance;
  // …

  public void deposit (double amount) {
    bankAccount.deposit (amount);
  }
  // …
}
```

Delegation and Polymorphism



Delegation and Polymorphism



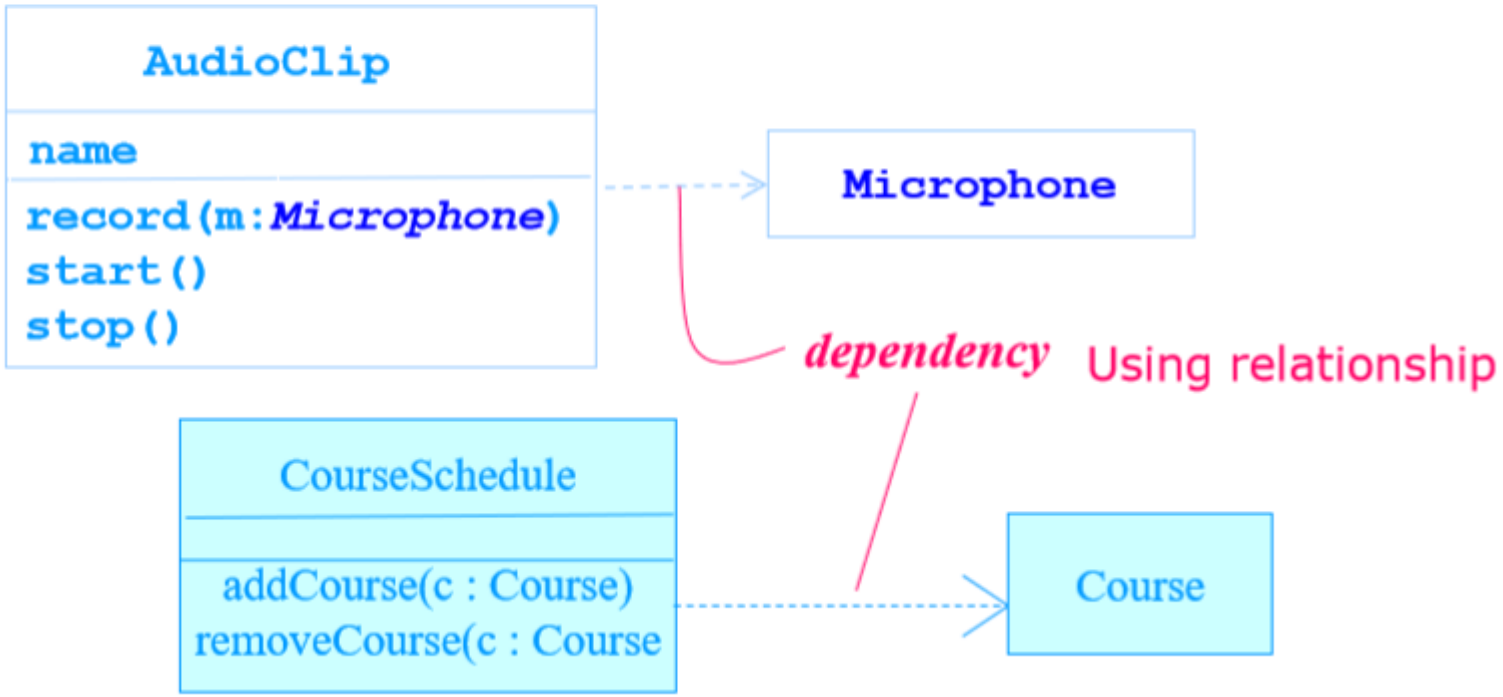
Inheritance Vs. Delegation

- Delegation => one more object in memory compared to inheritance
- Cost of this one more object = 4 bytes (the size of a reference variable)
- Extra level of indirection, resulting in slight performance drop
- Extra level of indirection, resulting in increased difficulty in debugging a program

- **Refer Module1_Demo 1**

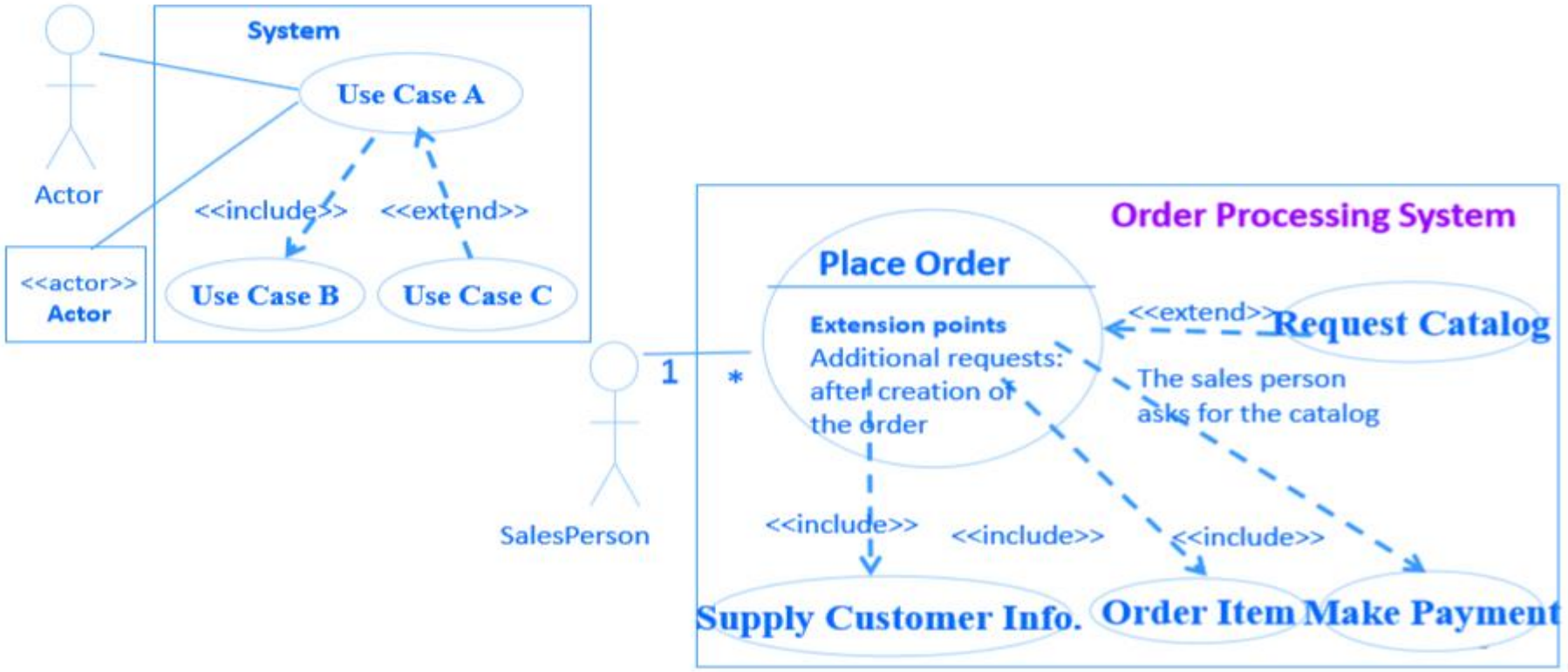
Dependency

- A change in one thing may affect another.
  - The most common dependency between two classes is one where one class <<use>>s another as a *parameter to an operation*.



Dependency among Use Cases

- Two Stereotypes of Dependency Among Use Cases:
  - *extend*: the target use case extends the behavior of the source
  - *include*: the source use case explicitly incorporates the behavior of another use case *at a location* specified by the source



SYNERGETICS GET IT RIGHT | Microsoft Partner
Gold Cloud Platform
Silver Learning

Module 2

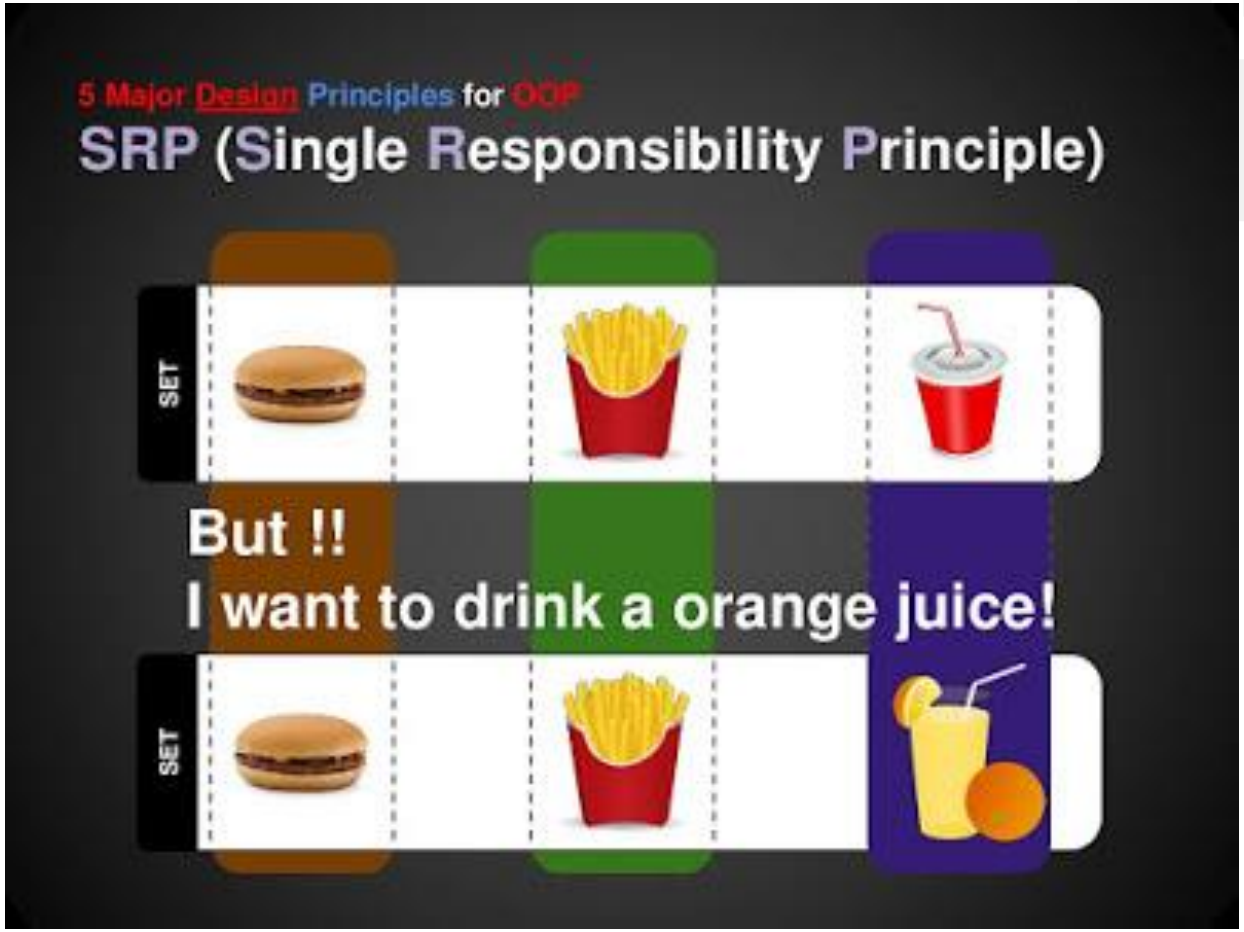
OOAD's SOLID principles

OOAD Principles

- (SRP) Single Responsibility Principle
- (OCP) Open-Closed Principle
- (LSP) Liskov Substitution Principle
- (ISP) Interface Segregation Principle
- (DIP) Dependency Inversion Principle

It’s said (Wikipedia) when all five principles are applied together intend to make it more likely that a programmer will create a system that is easy to maintain and extend over time.

SRP: Single Responsibility Principle



SRP: Single Responsibility Principle

"One class should have one and only one responsibility"

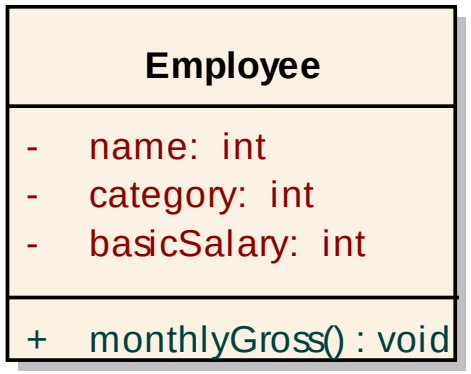
- A class should have a single responsibility, and that responsibility should be fully encapsulated in that class.
- Robert Martin: A class should have only one reason to change.
- Related to the principle of high cohesion. A class with low cohesion has multiple reasons to change.
- Example:

```
class Rectangle {
        getArea ()
        draw ()
}
```

  has low cohesion / multiple responsibilities.
- Better to have many small classes, rather than few large ones.

OCP: Open and Close Principle

"Software components should be open for extension, but closed for modification"




```
if (category == Employee.MANAGER)
    hra = 0.5 * basicSalary;
else // if (category == Employee.EXECUTIVE)
    hra = 0.25 * basicSalary;
return basicSalary + hra;
```

What if a new category, say, Clerk is added?

LSP: Liskov Substitution Principle

"Derived types must be completely substitutable for their base types"

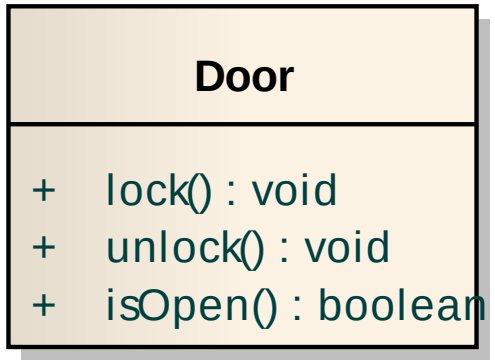
- The precondition of a derived class should not be stronger than the precondition of the base class, and the postcondition of a derived class should not be weaker than the postcondition of the base class.

ISP: Interface Segregation Principle

"Clients should not be forced to implement unnecessary methods which they will not use"

ISP: Interface Segregation Principle

"Clients should not be forced to implement unnecessary methods which they will not use"



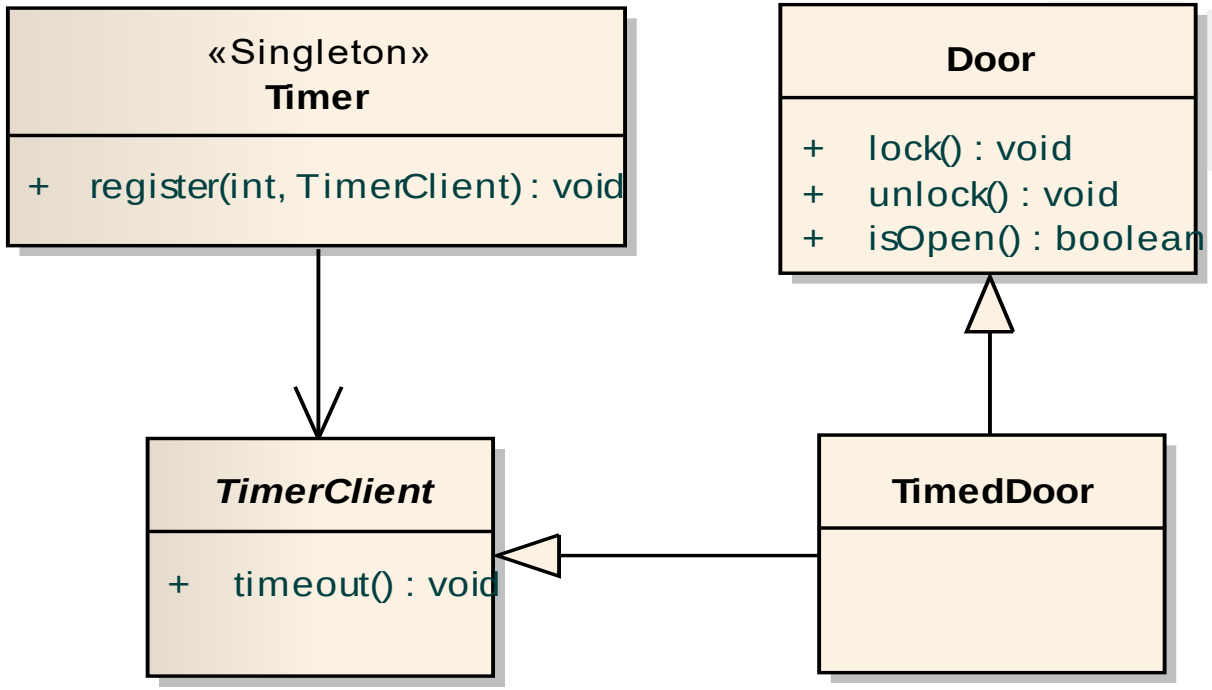
Now we need a door which needs to sound an alarm when it has been left open for too long.

Should we:

(1) add this behaviour in the Door class?

(2) subclass Door and add the behaviour in the subclass?

ISP: Interface Segregation Principle



ISP: Interface Segregation Principle

"Don't depend on things you don't need".

```
public interface Messenger
 {

askForCard();
tellInvalidCard();
askForPin();
tellInvalidPin(); tellCardWasSiezed();
askForAccount();
tellNotEnoughMoneyInAccount();
tellAmountDeposited(); tellBalance();

}
```

```
public interface LoginMessenger {
askForCard();
tellInvalidCard();
askForPin();
tellInvalidPin();
}

public interface WithdrawalMessenger
{
tellNotEnoughMoneyInAccount();
askForFeeConfirmation();
}
 publc class EnglishMessenger implements
LoginMessenger, WithdrawalMessenger
{ ... }
```

Liskov Substitution Principle —> (depends on) —-> Open for extension but closed for modification — > (depends on) —> Dependency Inversion Principle.

SYNERGETICS GET IT RIGHT | Microsoft Partner
Gold Cloud Platform
Silver Learning

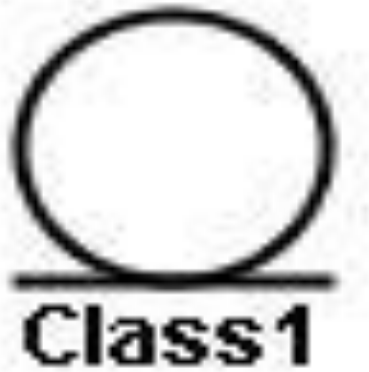
Module 3

Types of classes

Types of Classes

- Entity classes
- Boundary classes
- Data store classes
- Controller classes

Entity Classes



- Encapsulate the core requirements, business logic of an application.
- Also known as domain objects, representing the real-world domain.
- Layer containing these classes is often known as the “domain layer”.
- Guideline:
  - Avoid dependencies of entity classes on non-entity classes.

Boundary Classes

- Encapsulate the various interfaces of the system.
  - User interface classes
  - System interface classes
  - Device interface classes

Controller Classes

- Encapsulate the logic of individual use cases.
- Use them as mediators, to minimize dependencies between other classes.

Data Store Classes

- Move data between domain objects and a database while keeping the domain objects independent of the database.
- Encapsulate data storage and retrieval mechanisms.
- Also known as the persistence layer.

- Guideline:
  - Separate data access code, so that changes to database design have no impact on the remaining application.

Types of Classes

- Analysis classes are refined during design to become *entity classes*

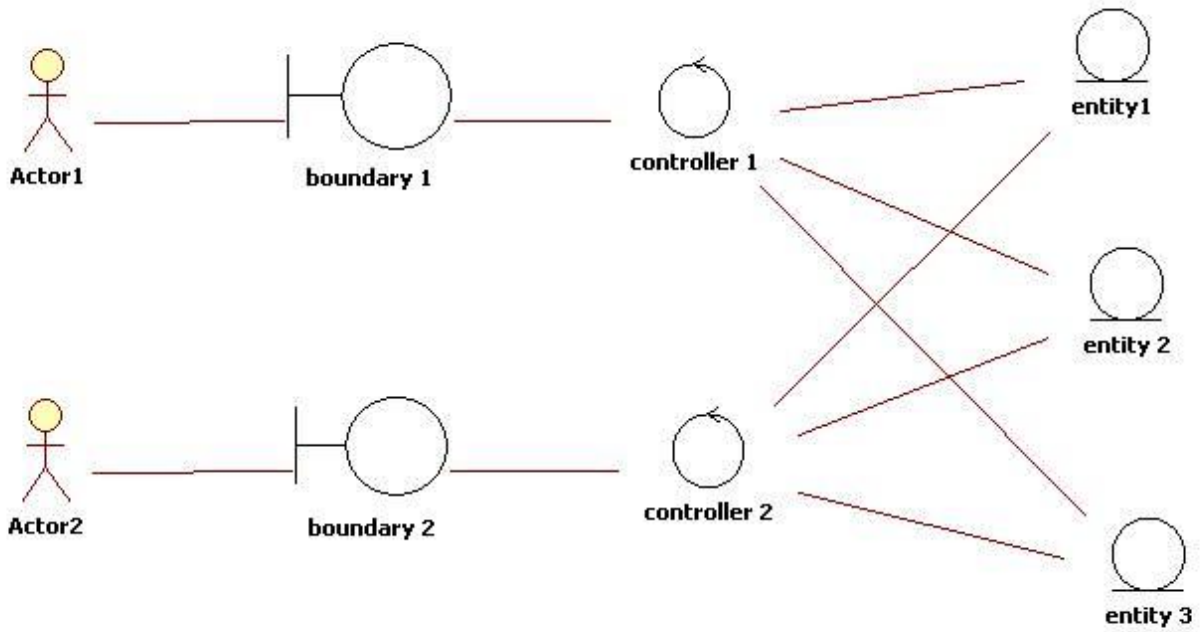
- Boundary classes are developed during design to create the interface (e.g interactive screen or printed reports) that the user sees and interacts with the software is used .
  - Boundary classes are designed with the responsibility of managing the way entity object are represented to users.

- Controller classes are designed to manage
  - The creation or update of entity objects.
  - The instantiation of boundary objects as they obtain information from entity objects.
  - Validation of data communication between objects or between the user and the application.

Types of Classes

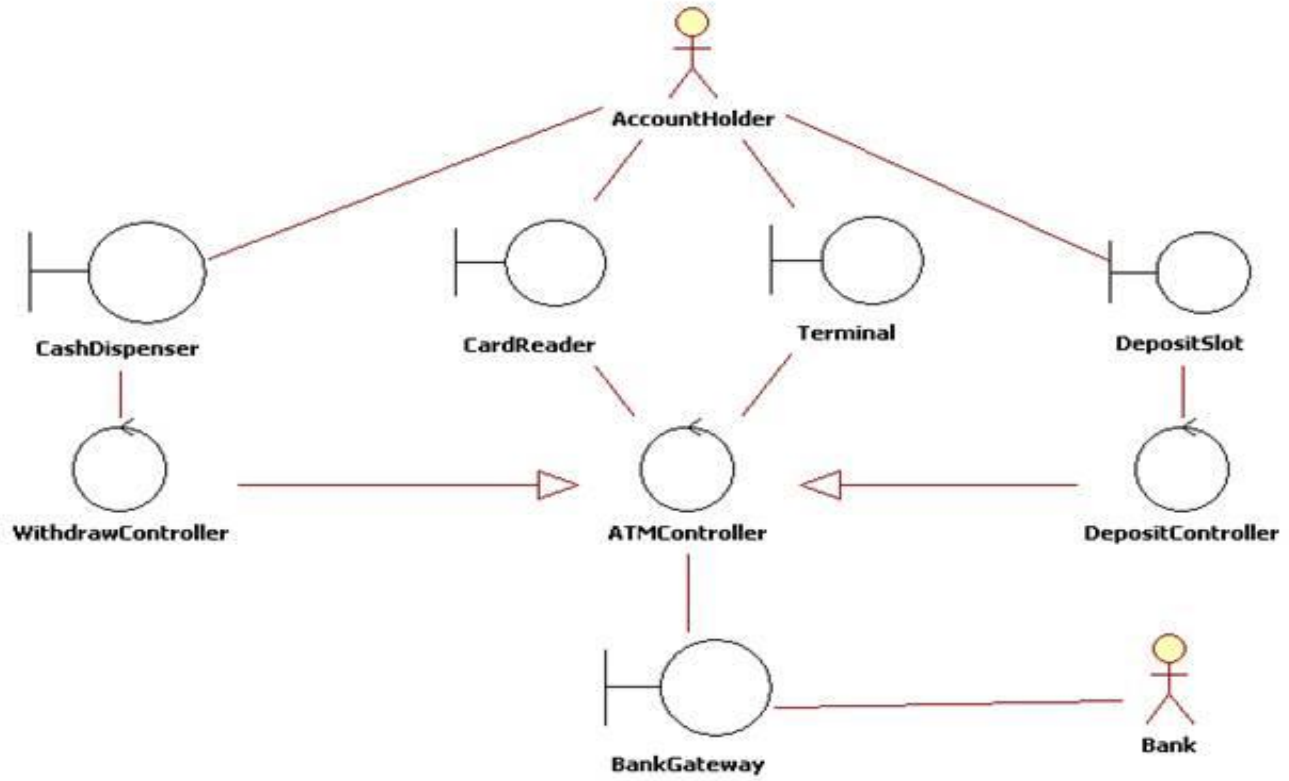
- Entity classes
- Boundary classes
- Data store classes
- Controller classes

The Entity-Control-Boundary Pattern



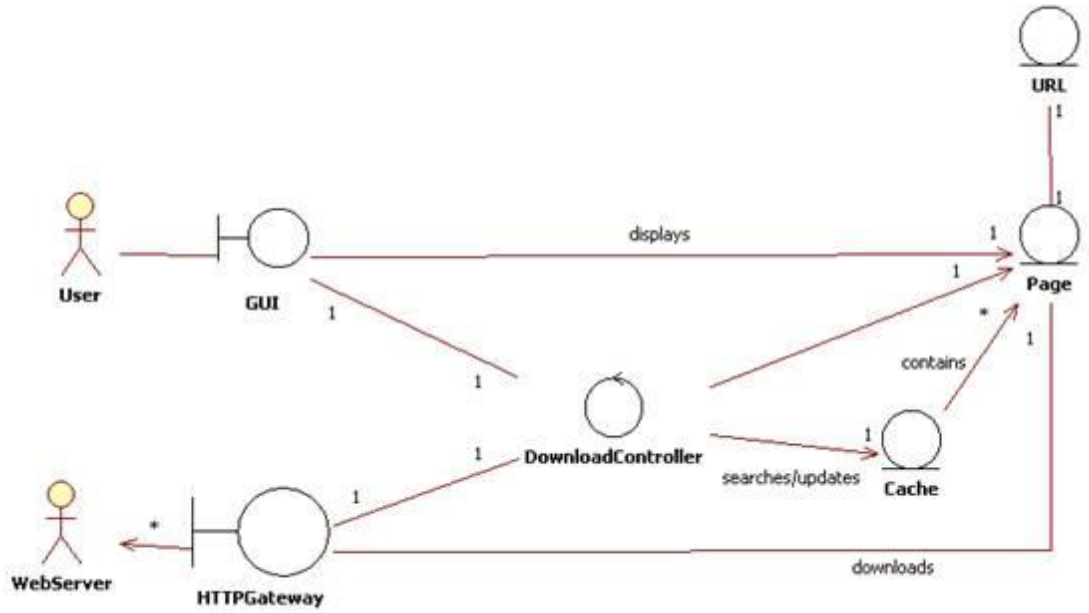
Types of Classes

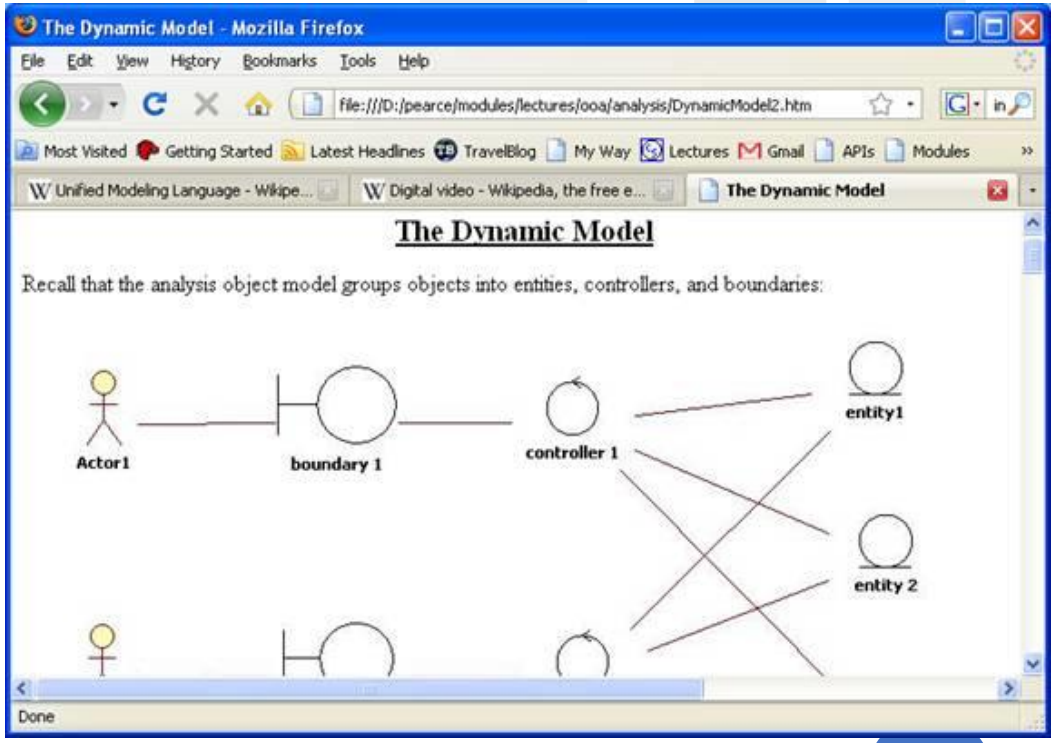
Analysis Model for a Simple Automatic Teller Machine (ATM)



Types of Classes

Analysis Model for a Simple Web Browser





SYNERGETICS GET IT RIGHT | Microsoft Partner
Gold Cloud Platform
Silver Learning

Module 4

Design patterns

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Module 4

i. Overview of design patterns?
ii. Creational, Structural and Behavioral patterns
iii. The factory pattern in brief.

What are design patterns?

- In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Why Design Patterns ?

1.Design Patterns are already defined and provides industry standard approach to solve a recurring problem, so it **saves time** if we sensibly use the design pattern.

2.Using design patterns promotes **reusability** that leads to more robust and **highly maintainable** code. It helps in **reducing total cost** of ownership (TCO) of the software product.

3.Since design patterns are already defined, it makes our **code easy to understand and debug.** It leads to **faster development** and new members of team understand it easily.

Importance of selecting the Correct Design Pattern

- A design pattern is not a class or a library that we can simply plug into our system; it's much more than that.
- It is a template that has to be implemented in the correct situation. It's not language-specific either.
- Most importantly, any design pattern can be a double-edged sword— if implemented in the wrong place, it can be disastrous and create many problems for you. However, implemented in the right place, at the right time, it can be your savior.

Three basic kinds of design patterns

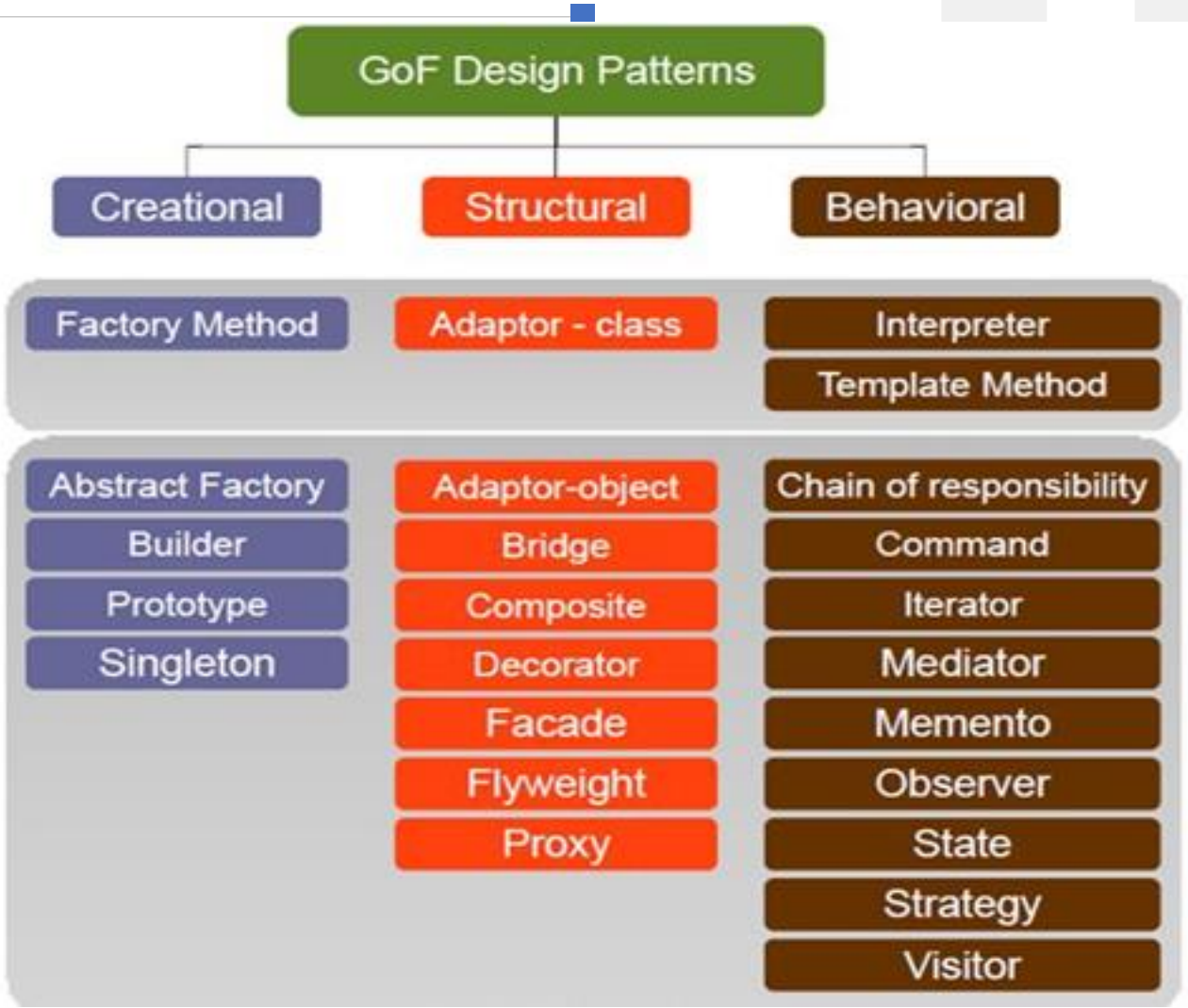
- There are three basic kinds of design patterns:
  - **structural**
  - **creational**
  - **Behavioral**

- **Structural** patterns generally deal with relationships between entities, making it easier for these entities to work together.

- **Creational** patterns provide instantiation mechanisms, making it easier to create objects in a way that suits the situation.

- **Behavioral** patterns are used in communications between entities and make it easier and more flexible for these entities to communicate.

GoF Design Patterns



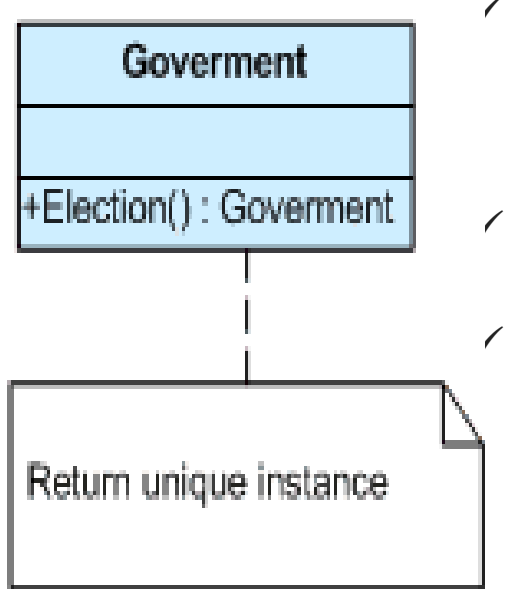
Creational Patterns

Singleton Abstract Factory

Creational Patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate

Singleton Pattern

Ensure that only one instance of a class is created and **Provide a global access point to the object.**



- ✓ **when** we must ensure that **only one instance of a class** is created and when the instance must be available through all the code.
- ✓ **when** multiple threads must access the same resources through the same singleton object.

- ✓ **Situations** when singleton pattern is used:

- ✓ - **Logger Classes**
  - Configuration Classes
  - Accessing resources in shared mode

Singleton Pattern

Advantage of Singleton design pattern

Saves memory because object is not created at each request. Only single instance is reused again and again.

There are **two forms** of singleton design pattern

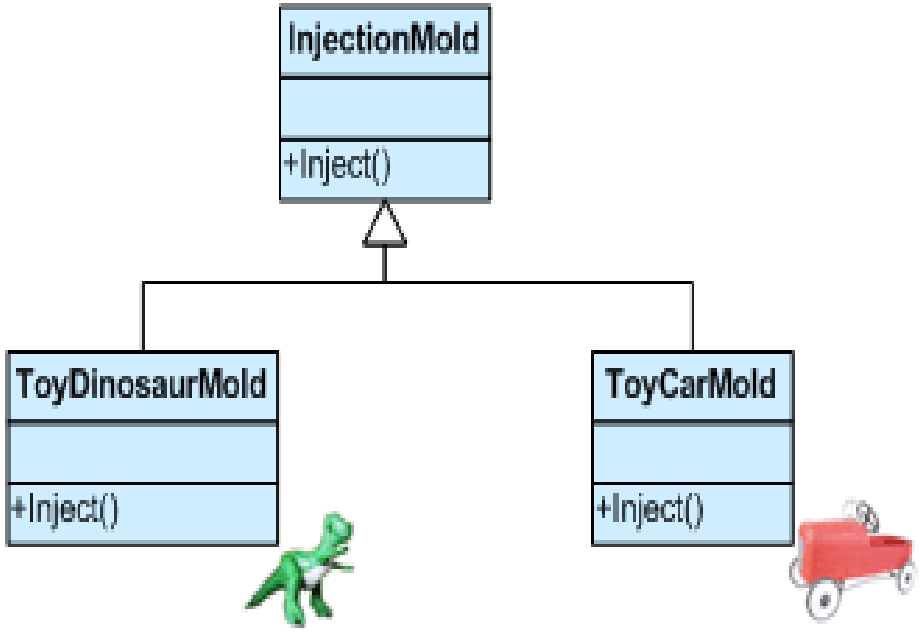
- Early Instantiation: creation of instance at load time.
- Lazy Instantiation: creation of instance when required.

Guidelines to implement Singleton pattern:

- ✓ Declare a private constructor for the class for which you want to make Singleton.
- ✓ Declare a static variable for that class.
- ✓ Create a static method for that class and assign the object to the declared static variable.

Factory Pattern

Creates objects without exposing the instantiation logic to the client and **Refers to the newly created object through a common interface.**



- ✓ **when:** - a framework delegate the creation of objects derived from a common superclass to the factory - we need flexibility in adding new types of objects that must be created by the class
- ✓ **Situations** when factory pattern is used:
- ✓ factories providing an xml parser: - javax.xml.parsers.DocumentBuilderFactory or javax.xml.parsers.SAXParserFactory
  - java.net.URLConnection - allows users to decide which protocol to use

Factory , Factory Method and Abstract Factory

- **Factory -** Creates objects without exposing the instantiation logic to the client and Refers to the newly created object through a common interface. *Is a simplified version of Factory Method*

- **Factory Method** - Defines an interface for creating objects, but let subclasses to decide which class to instantiate and Refers to the newly created object through a common interface.

- **Abstract Factory** - Offers the interface for creating a *family of related objects*, without explicitly specifying their classes.

When to use which?

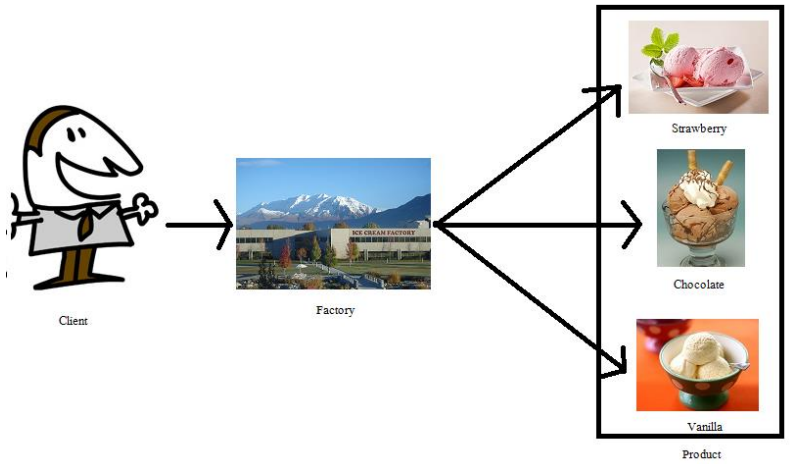
Factory: Client just need a class and does not care about which concrete implementation it is getting.

Factory Method: Client doesn't know what concrete classes it will be required to create at runtime, but just wants to get a class that will do the job.

AbstactFactory: When your system has to create multiple *families of products* or you want to provide a library of products without exposing the implementation details.
Abstract Factory classes are often implemented with Factory Method

Factory Method

Defines an interface for creating objects, but let subclasses to decide which class to instantiate and **Refers to the newly created object through a common interface.**



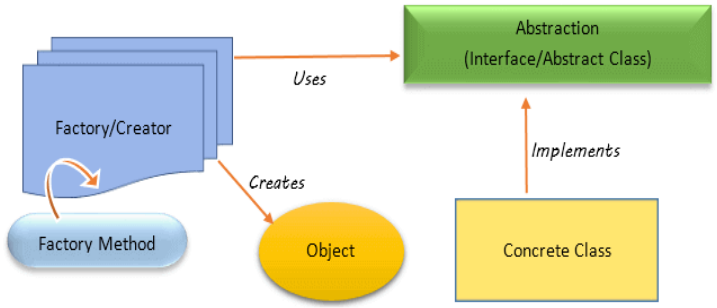
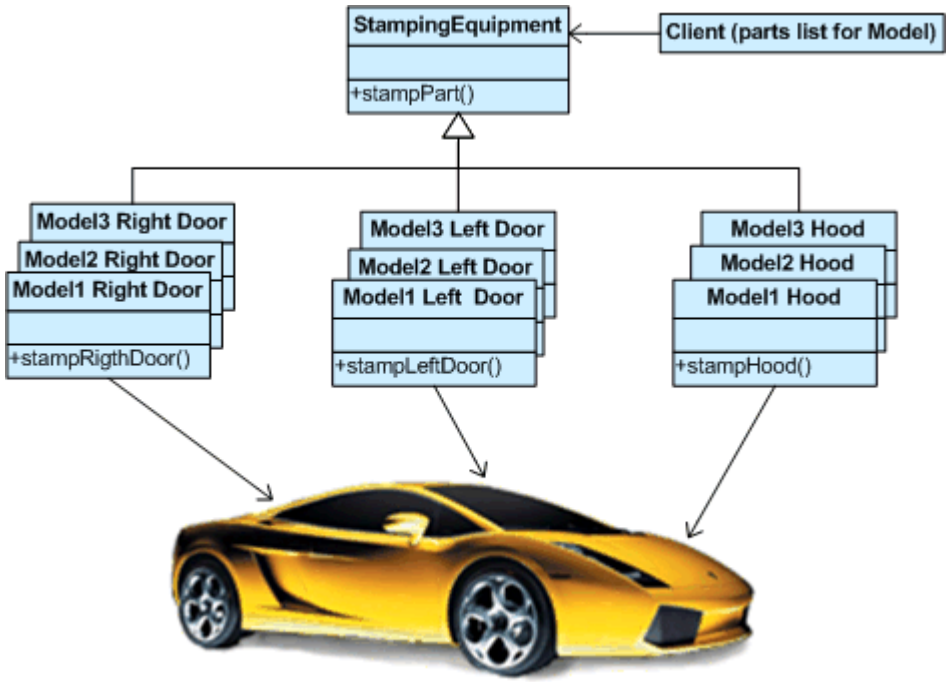


Figure 1 - Factory Method Overview

- ✓ **when:** - a framework delegate the creation of objects derived from a common superclass to the factory
  - the base factory class does not know what concrete classes will be required to create - delegates to its subclasses the creation of concrete objects
  - factory subclasses subclasses are aware of the concrete classes that must be instantiated
- ✓ **Situations** when factory Method pattern is used:
- ✓
- ✓ factories providing an xml parser: javax.xml.parsers.DocumentBuilderFactory or javax.xml.parsers.SAXParserFactory

Abstract Factory

Offers the interface for creating a family of related objects, without explicitly specifying their classes.



when:

- ✓ A system should be configured with one of multiple families of products .A system should be independent of how its products are created, composed and represented

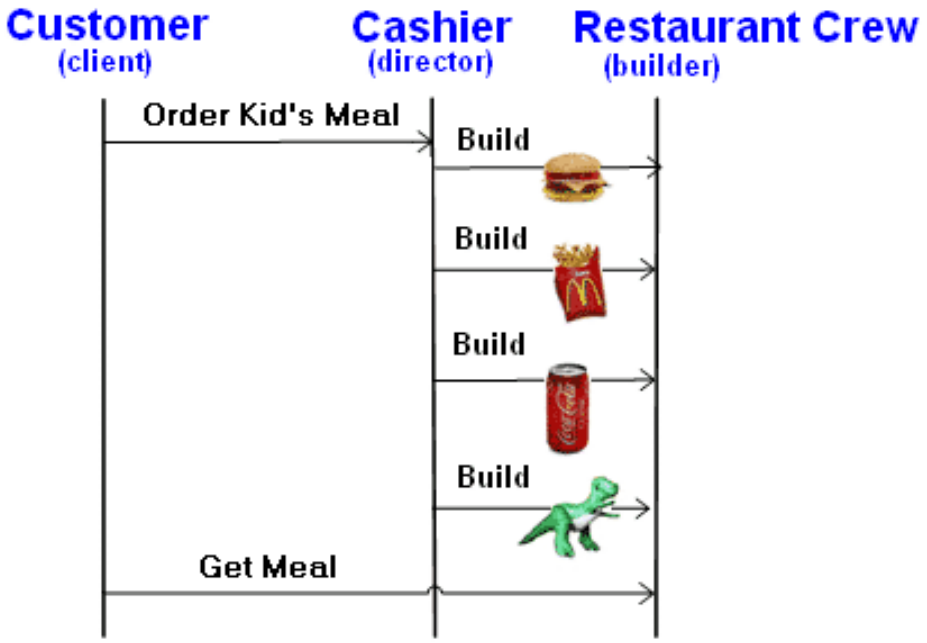
Products from the same family should be used all together, products from different families ahould not be used togheter and this constraint must be ensured.Only the product interfaces are revealed, the implementations remains hidden to the clients.

Situations:

- ✓ Java.awt.Toolkit , javax.swing.LookAndFeel
- ✓ java.sql.Connection - an abstract factory which create Statements, PreparedStatements, CallableStatements,... for each database flavor.

Builder

Defines an instance for creating an object but letting subclasses decide which class to instantiate and Allows a finer control over the construction process.



- ✓ **when:** - An application might need a mechanism for building complex objects that is independent from the ones that make up the object. The creation algorithm of a complex object is independent from the parts that actually compose the object
  the system needs to allow different representations for the objects that are being built.

- ✓ **Situations**
- ✓ implementations of java.lang.Appendable are infact good example of use of Builder pattern in java. e.g.
- ✓ java.lang.StringBuilder#append() [Unsynchronized class]
- ✓ java.lang.StringBuffer#append() [Synchronized class]

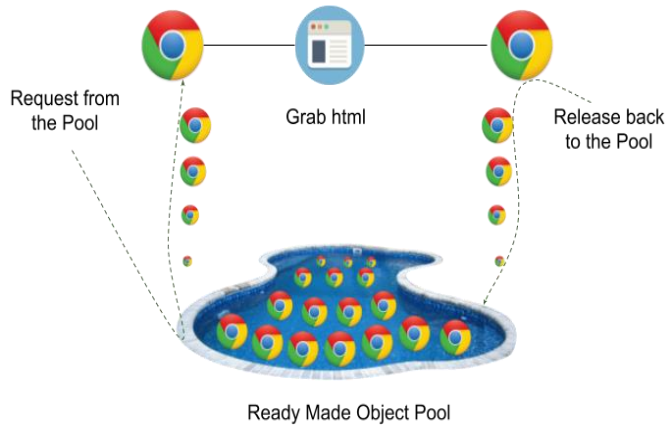
Object Pool

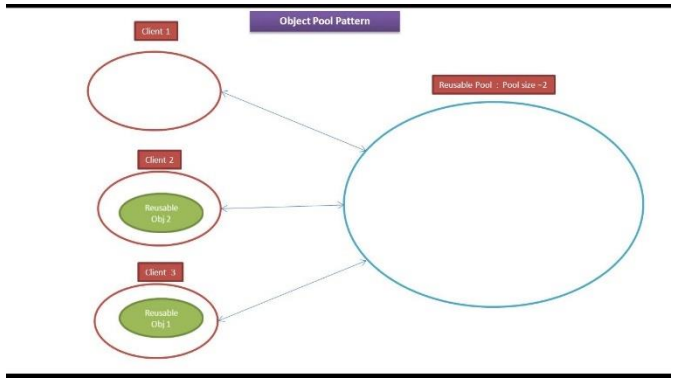
Reuses and shares objects that are expensive to create.

- ✓ **when:**
- ✓ whenever there are several clients who needs the same stateless resource which is expensive to create.

- ✓ **Situations** when Object pool pattern is used:
- ✓ The most common situations when object pool pattern is used:
  - Database Connections
  - Remote Objects

Object Pool





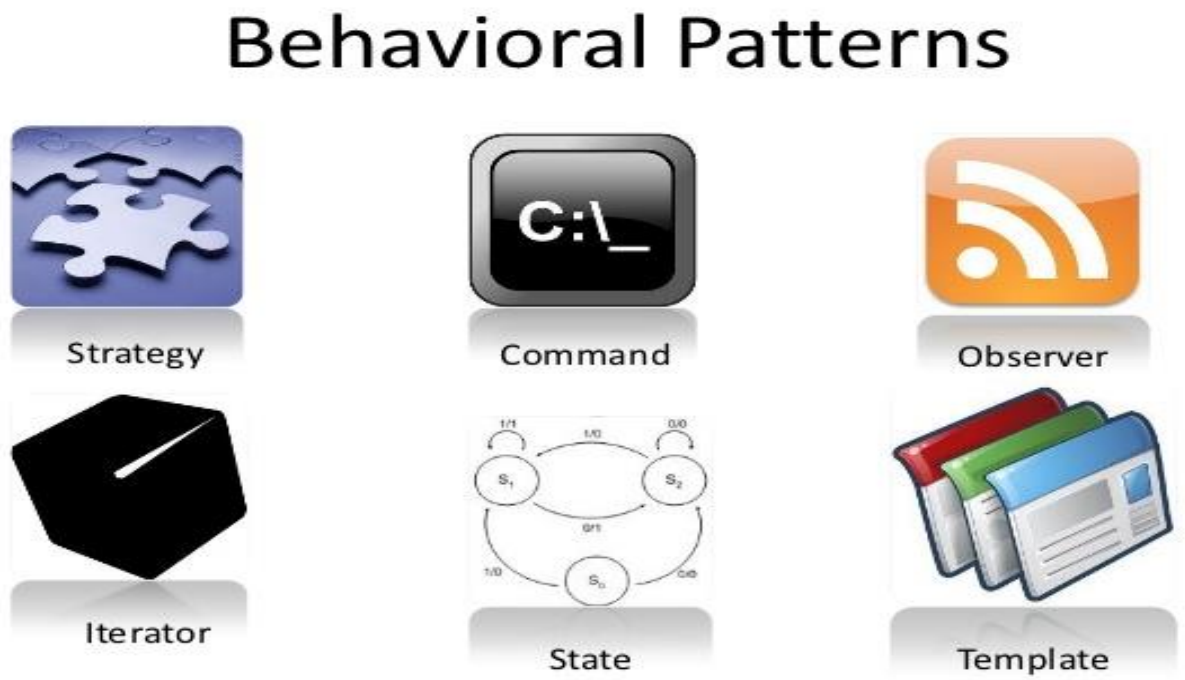
Without Object Pool :

Fire up a fresh browser instance - Very Slow
Navigate to the desired URL
Capture the html source
Quit or Destroy the browser instance - Very Slow
Return the HTML

With Object Pool :

Get an instance of browser from the Browser pool - Very fast
Navigate to the desired URL
Capture the html source
Release the browser instance back to the Browser Pool - Very fast
Return the HTML

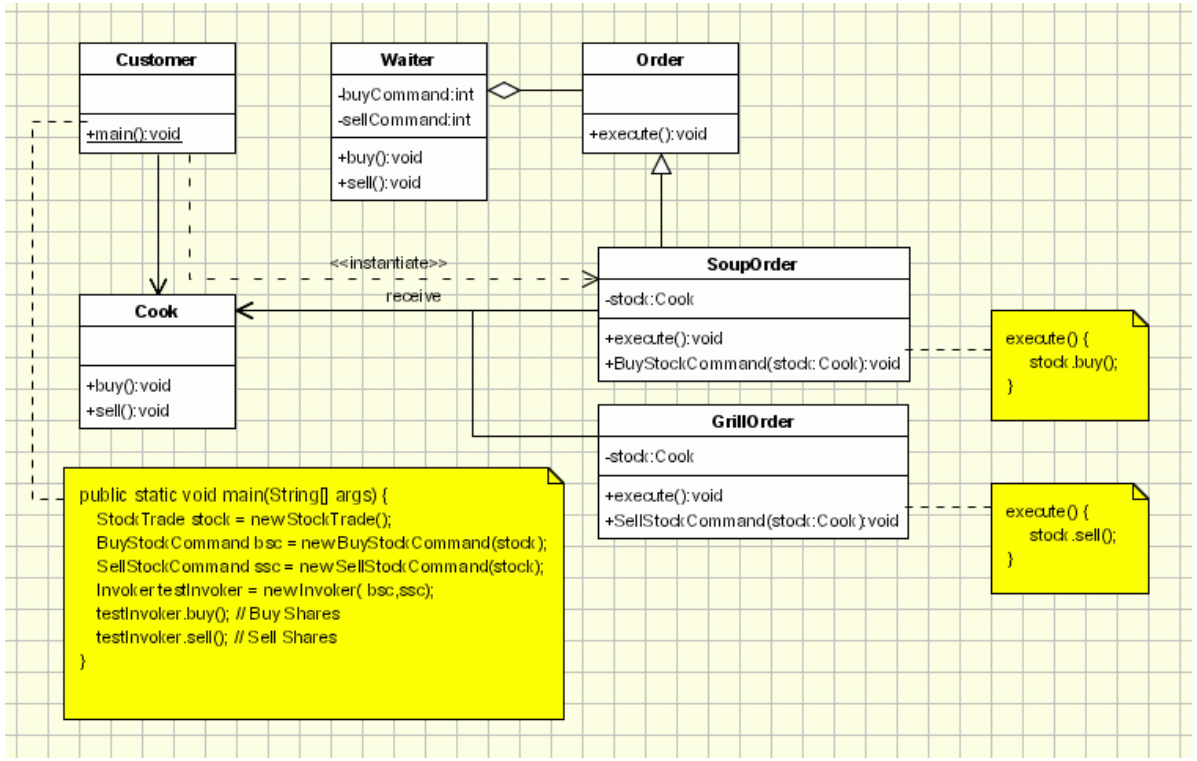
Behavioral Design Patterns:



Behavioral Patterns: concerned with how classes and objects interact and distribute responsibility

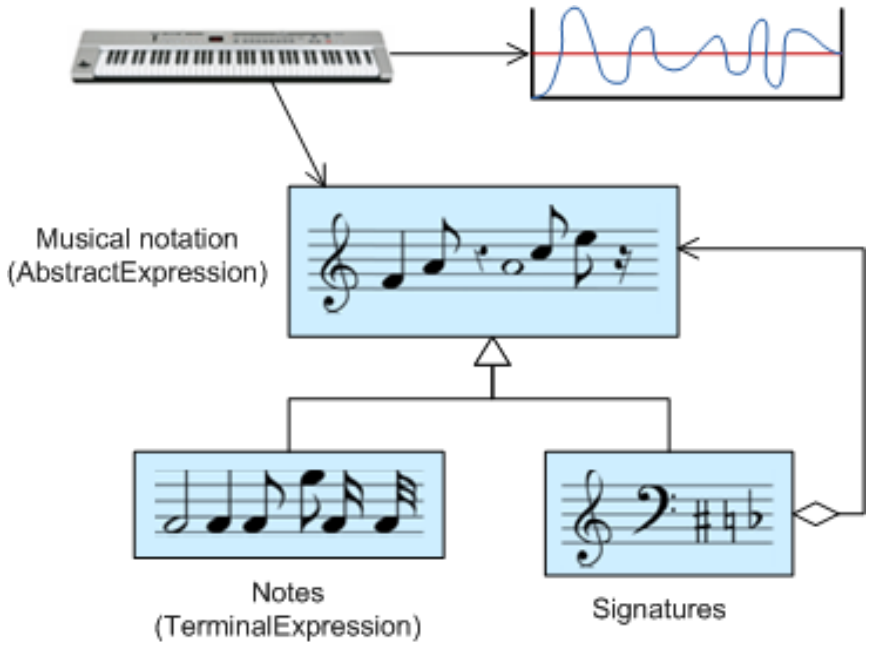
Command

Encapsulate a request in an object, Allows the parameterization of clients with different requests and Allows saving the requests in a queue.



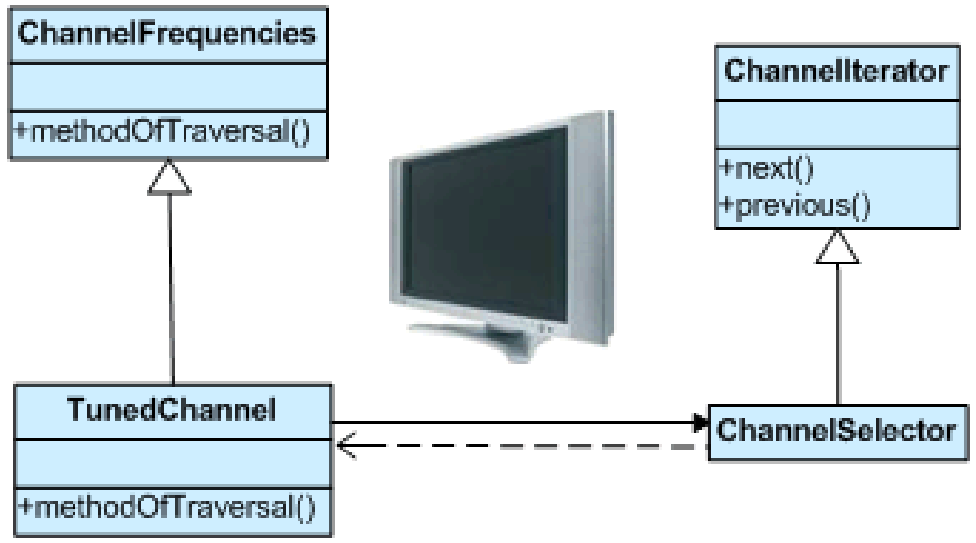
Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language / Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design



Iterator

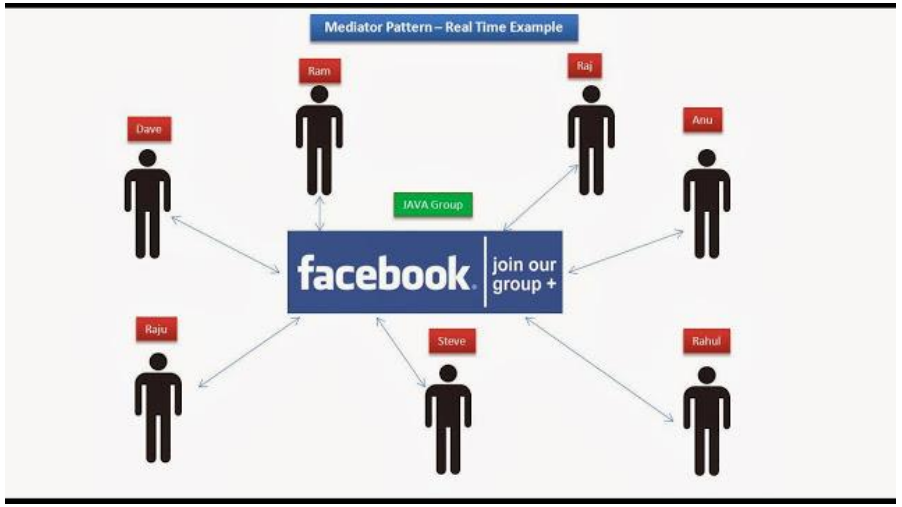
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



- Iterator design pattern hides the actual implementation of traversal through the collection and client programs just use iterator methods.
- The iterator pattern can be implemented in a customized way in Java according to need.
- We can use several iterators on the same collection.

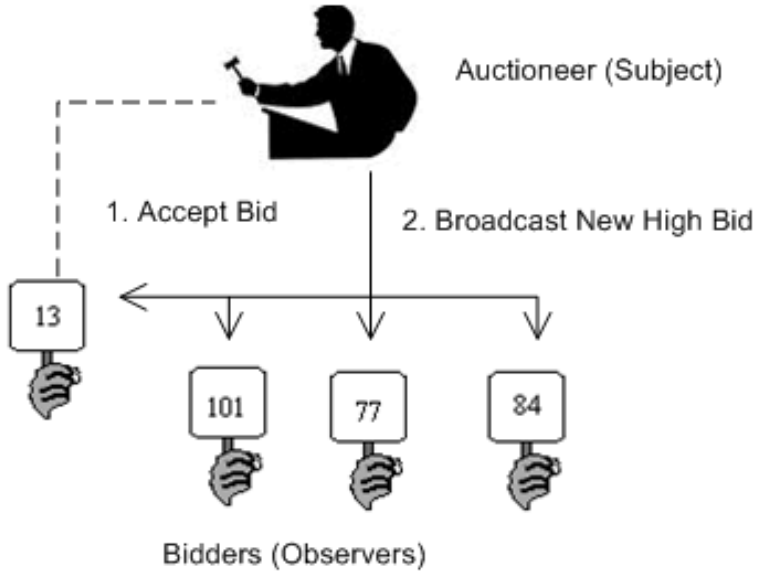
Mediator

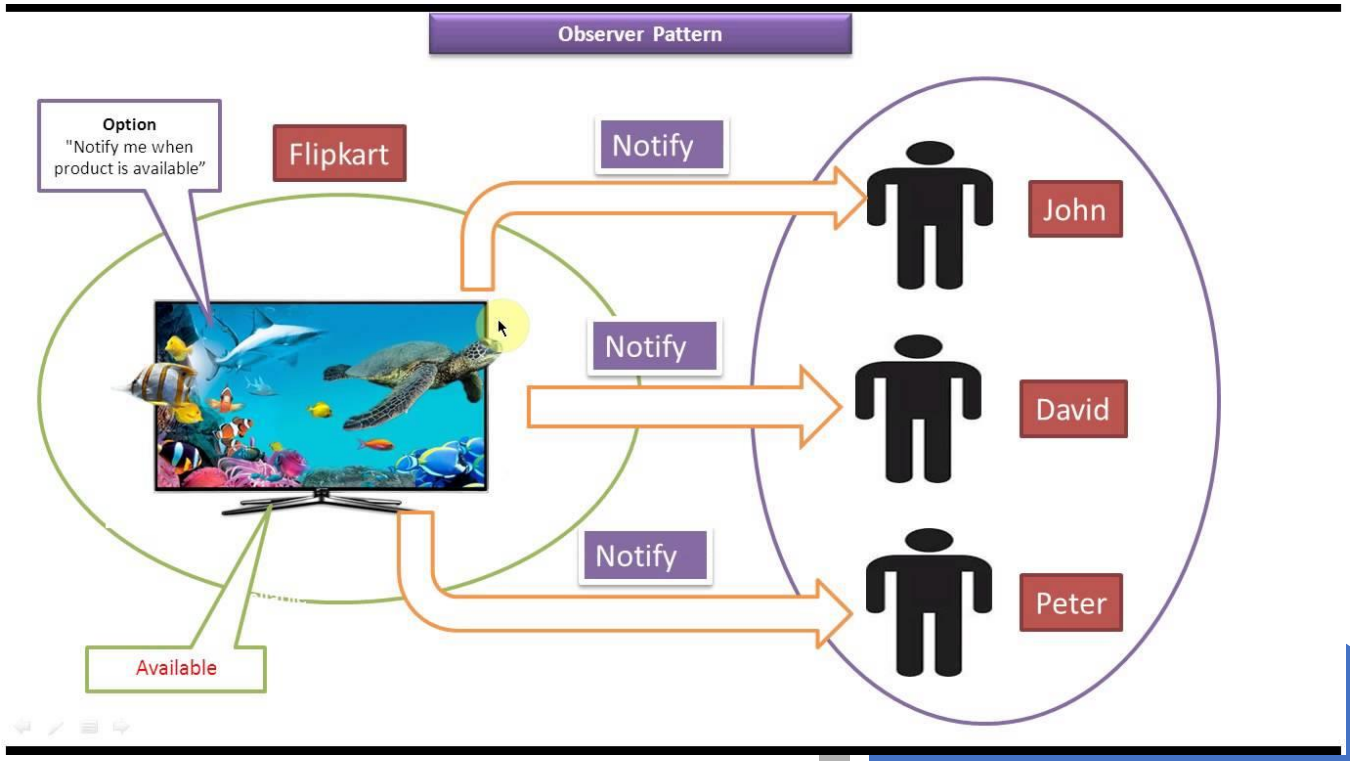
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.



Observer

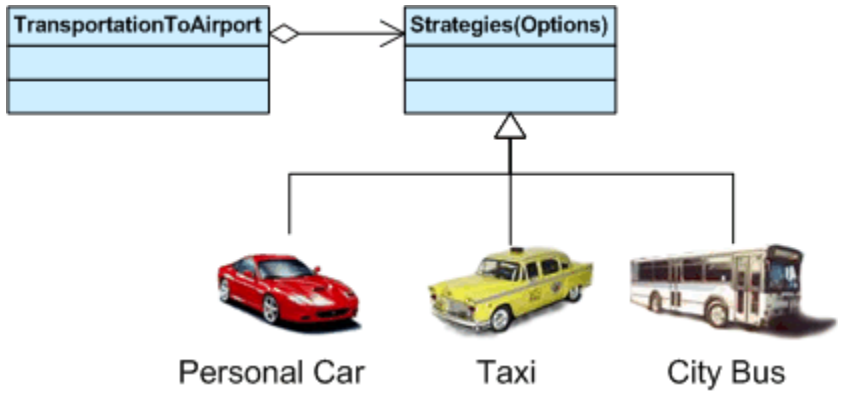
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

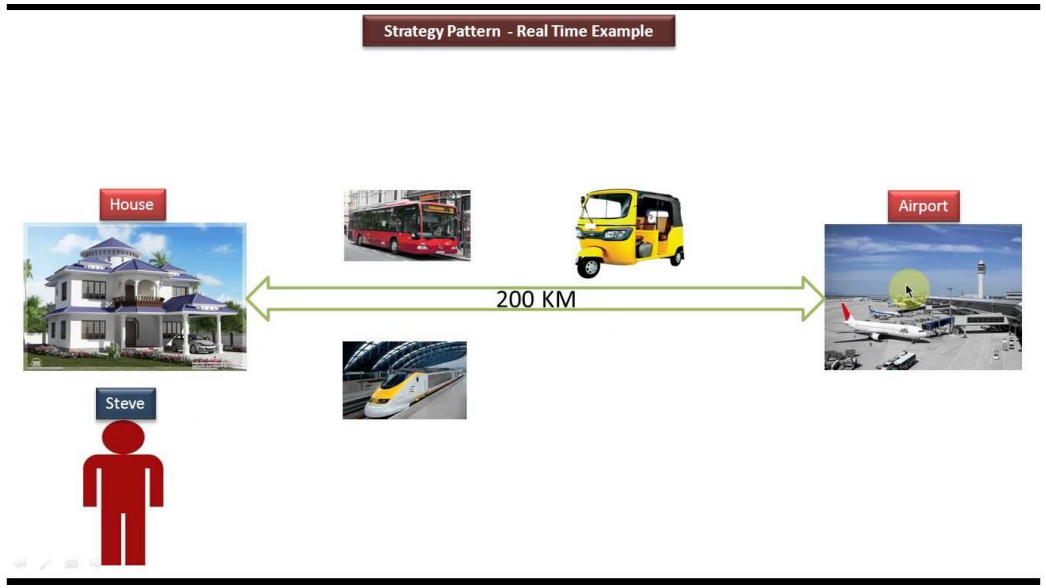




Strategy

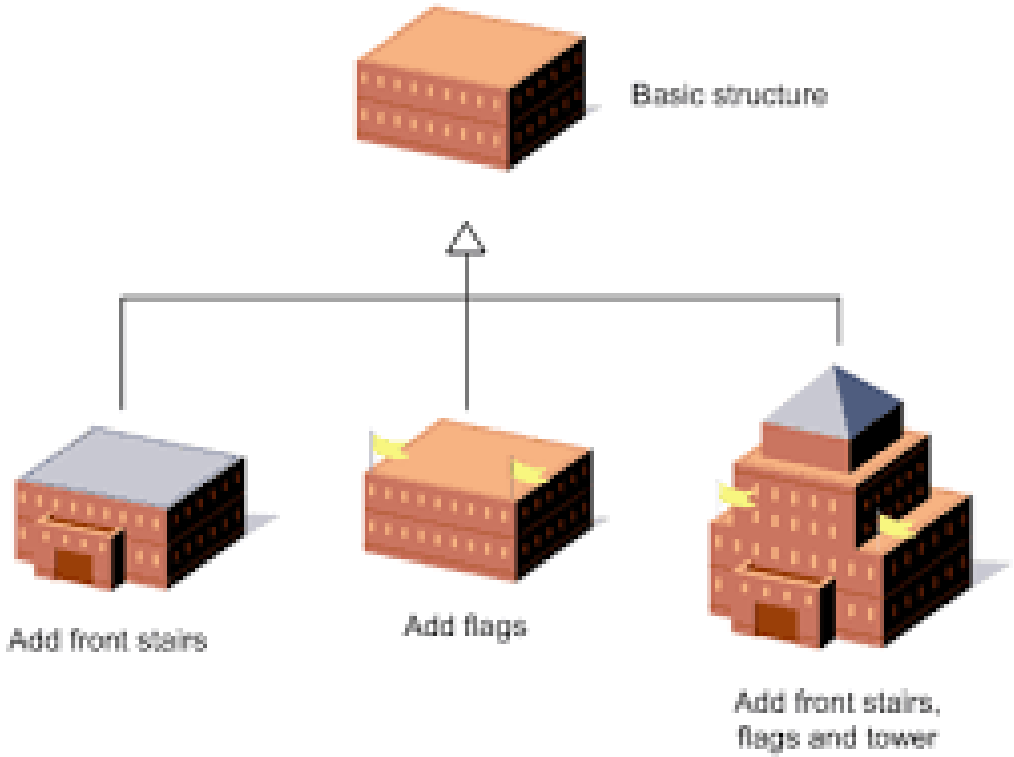
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.





Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses / Template Method lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure.



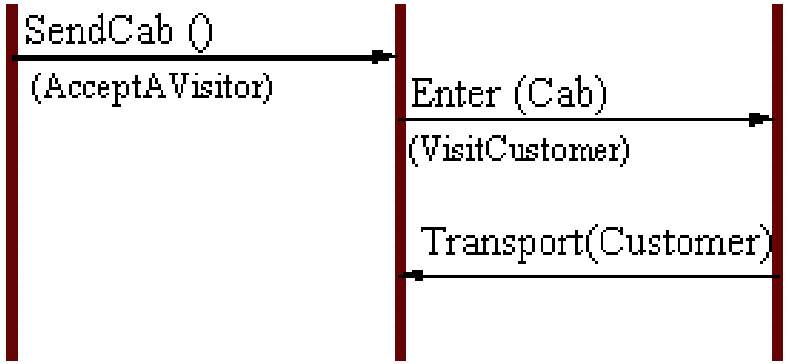
Visitor

Represents an operation to be performed on the elements of an object structure / Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Cab Company Dispatcher
(Object Structure is List of Customers)

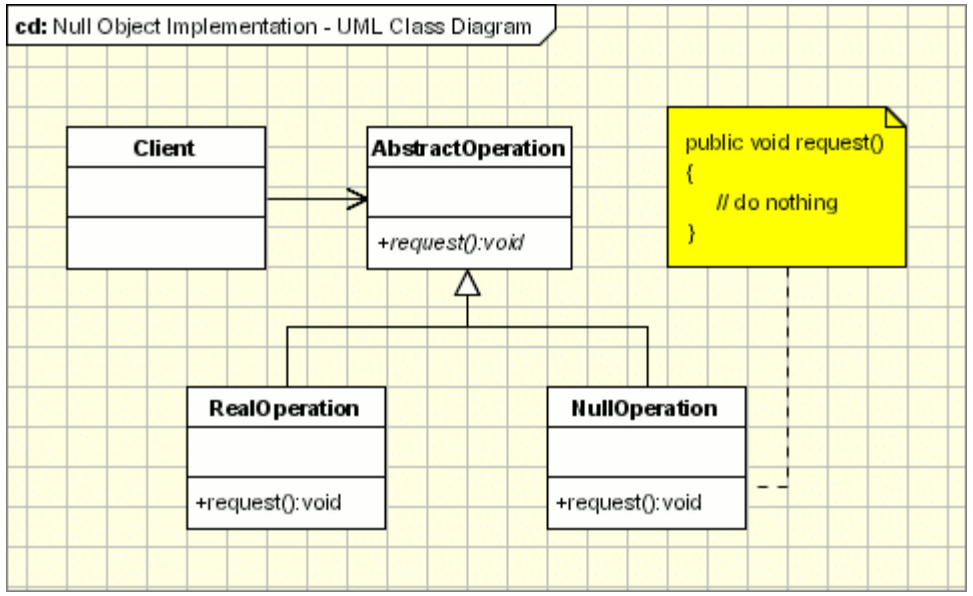
Customer
(Concrete Element of Customer List)

Taxi
(Visitor)



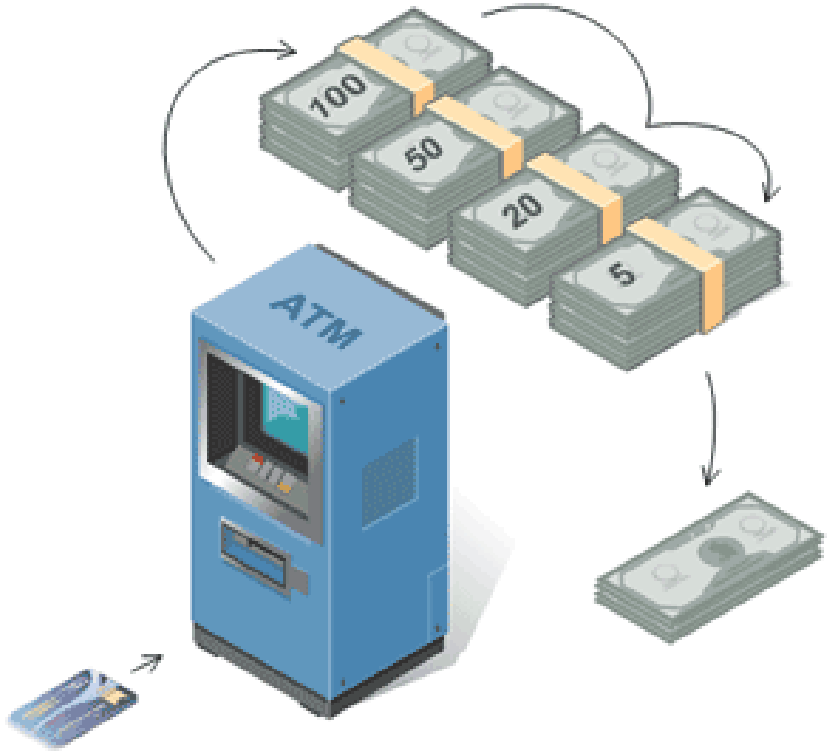
Null Object

Provide an object as a surrogate for the lack of an object of a given type. / The Null Object Pattern provides intelligent do nothing behavior, hiding the details from its collaborators.



Chain of Responsibiliy

The objects become parts of a chain and the request is sent from one object to another across the chain until one of the objects will handle it. Avoids attaching the sender of a request to its receiver, giving this way other objects the possibility of handling the request too.



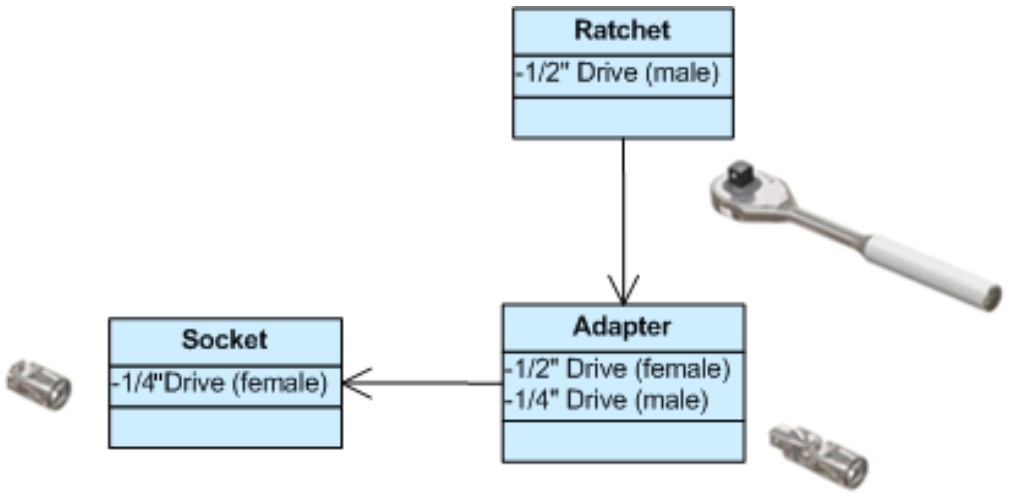
Structural pattern

Structural Patterns

Structural Patterns let you compose classes or objects into larger structures

Adapter

Convert the interface of a class into another interface clients expect. / Adapter lets classes work together, that could not otherwise because of incompatible interfaces.



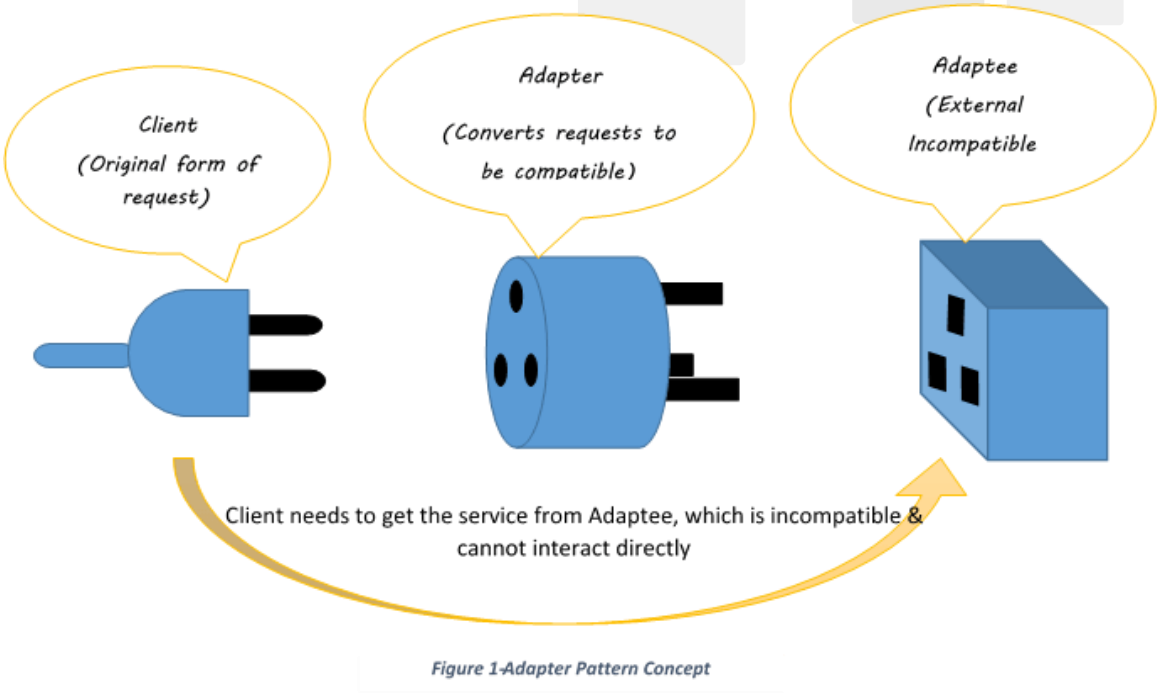
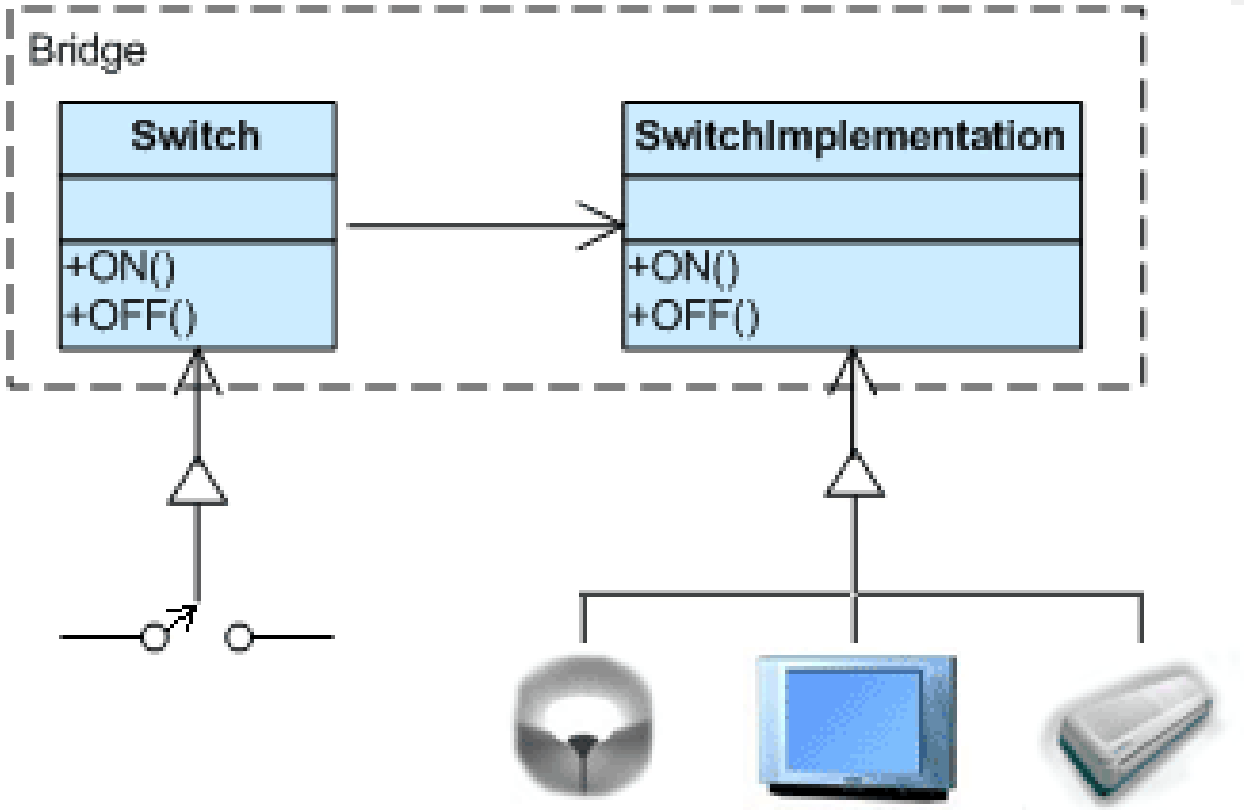


Figure 1-Adapter Pattern Concept

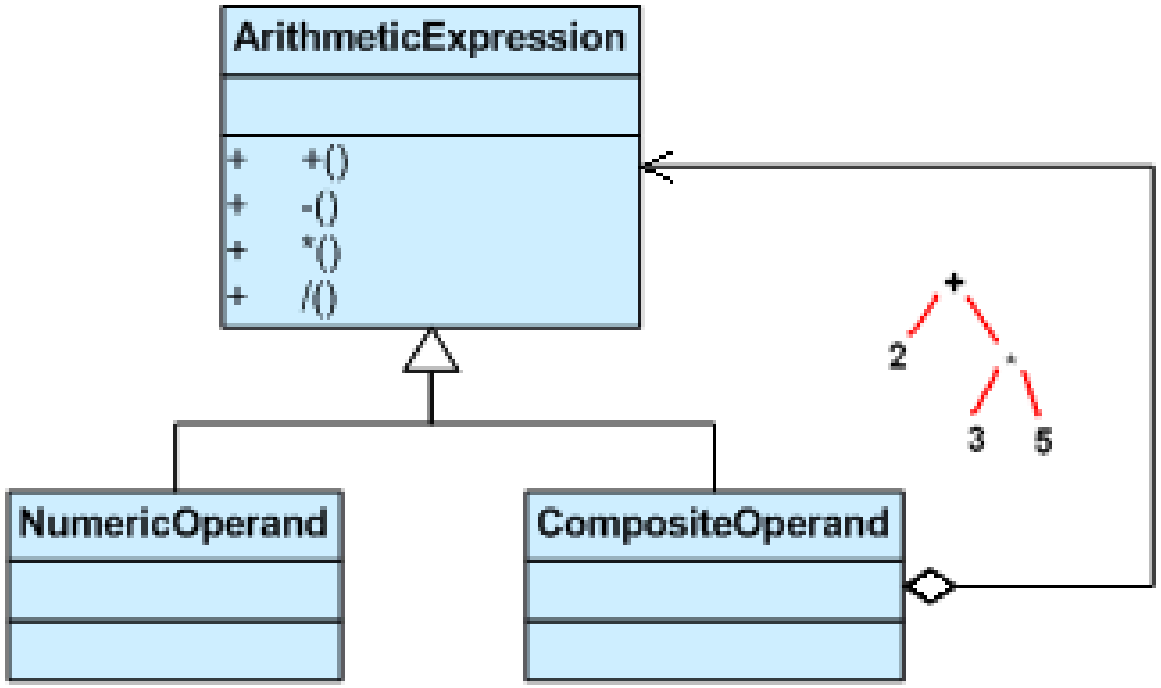
Bridge

Compose objects into tree structures to represent part-whole hierarchies. / Composite lets clients treat individual objects and compositions of objects uniformly.



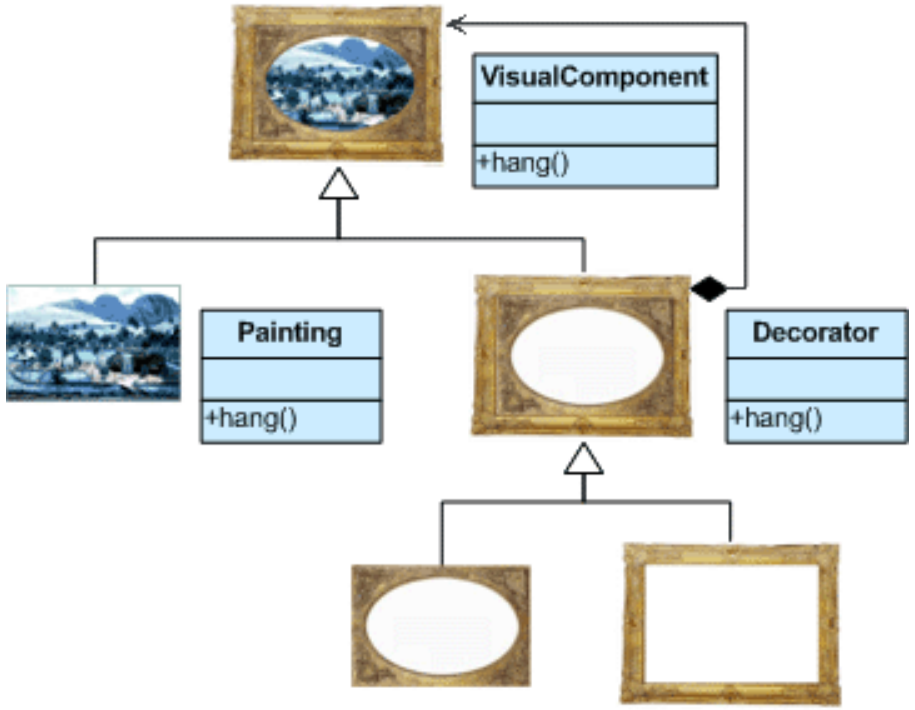
Composite

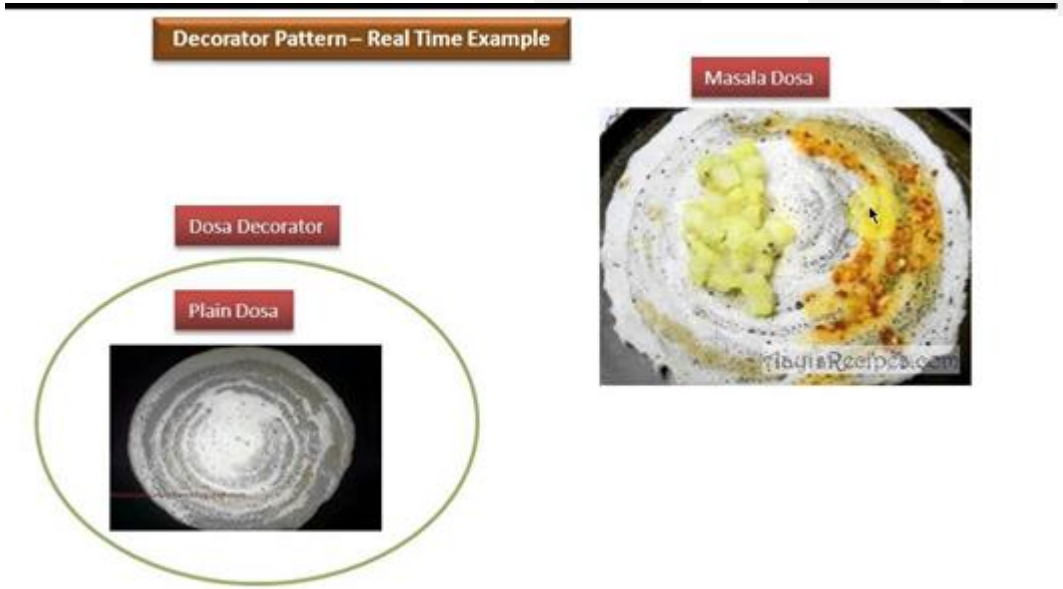
Compose objects into tree structures to represent part-whole hierarchies. / Composite lets clients treat individual objects and compositions of objects uniformly.



Decorator

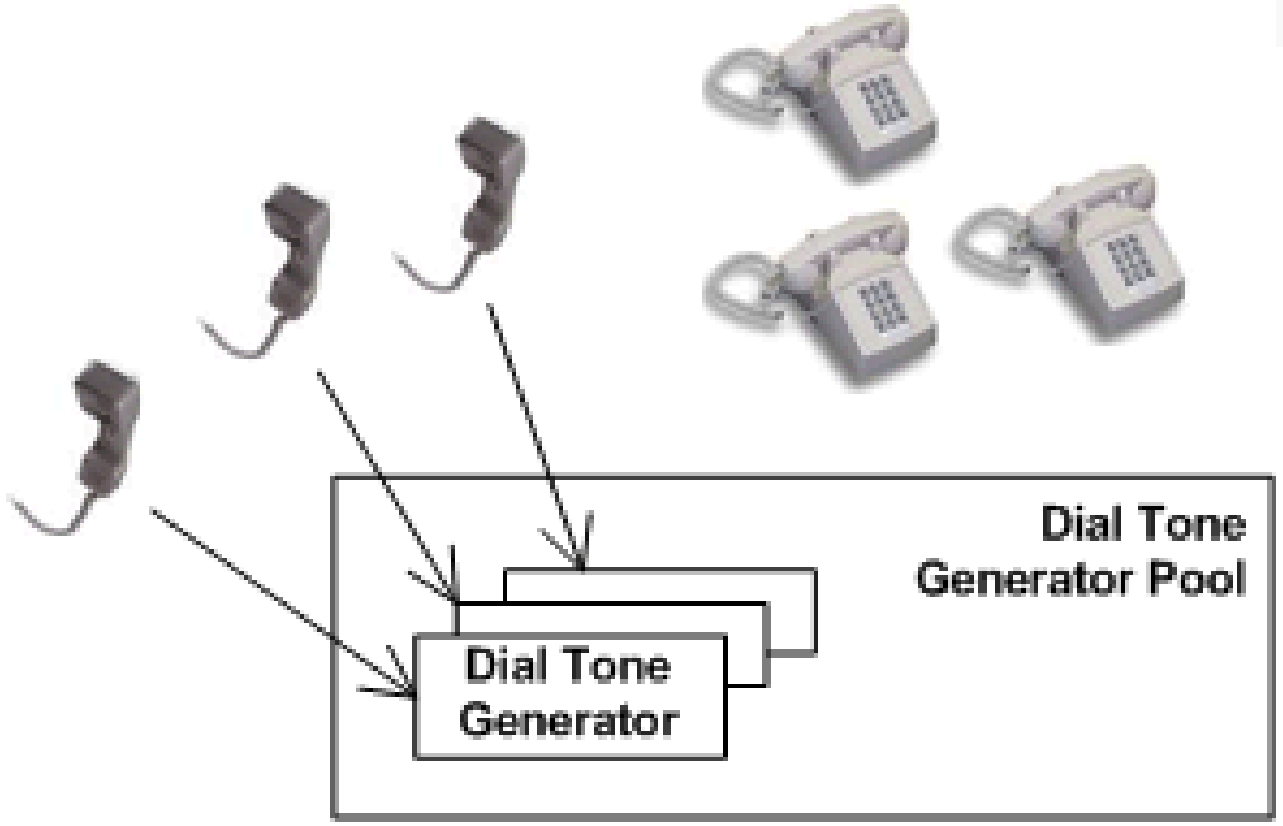
Add additional responsibilities dynamically to an object.





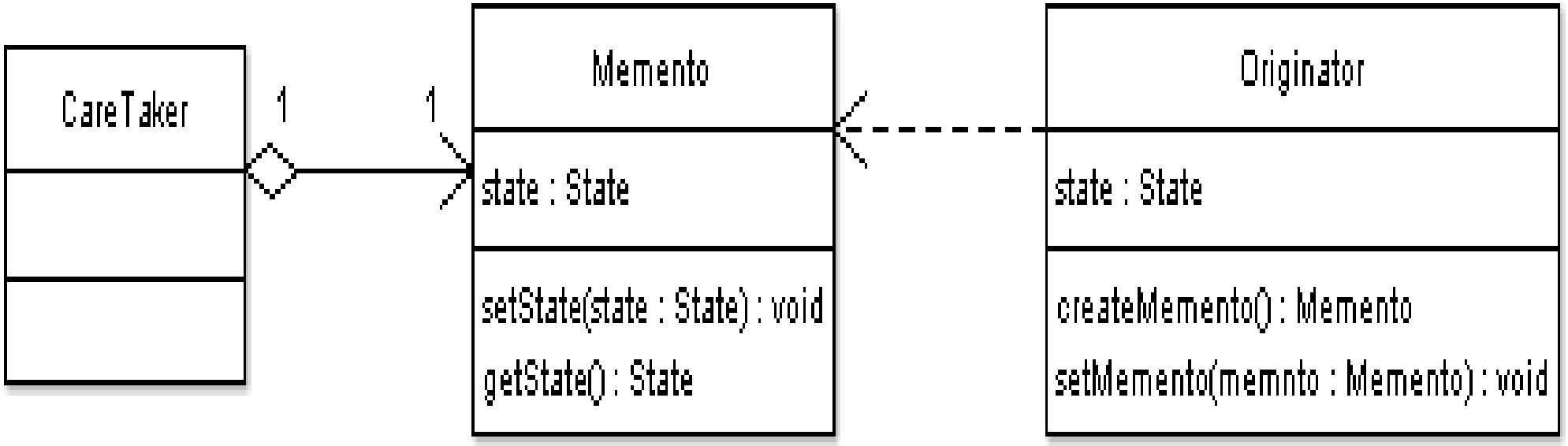
Flyweight

Use sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary.

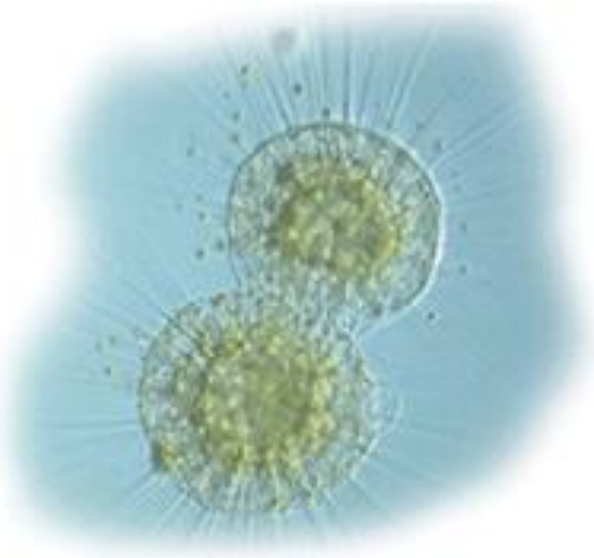


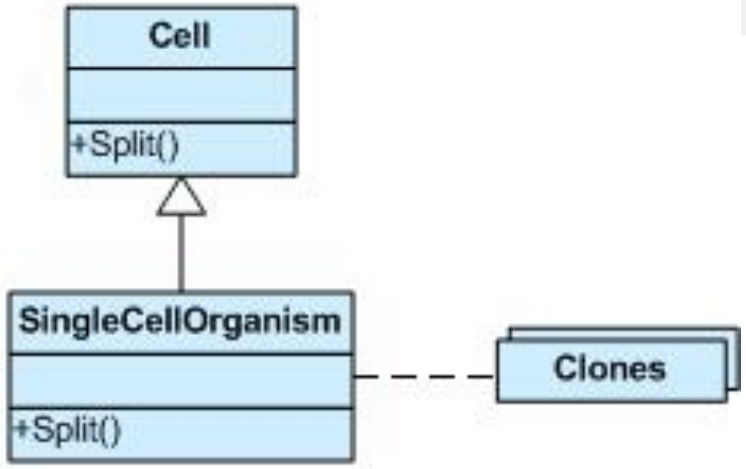
Memento

Capture the internal state of an object without violating encapsulation and thus providing a mean for restoring the object into initial state when needed.



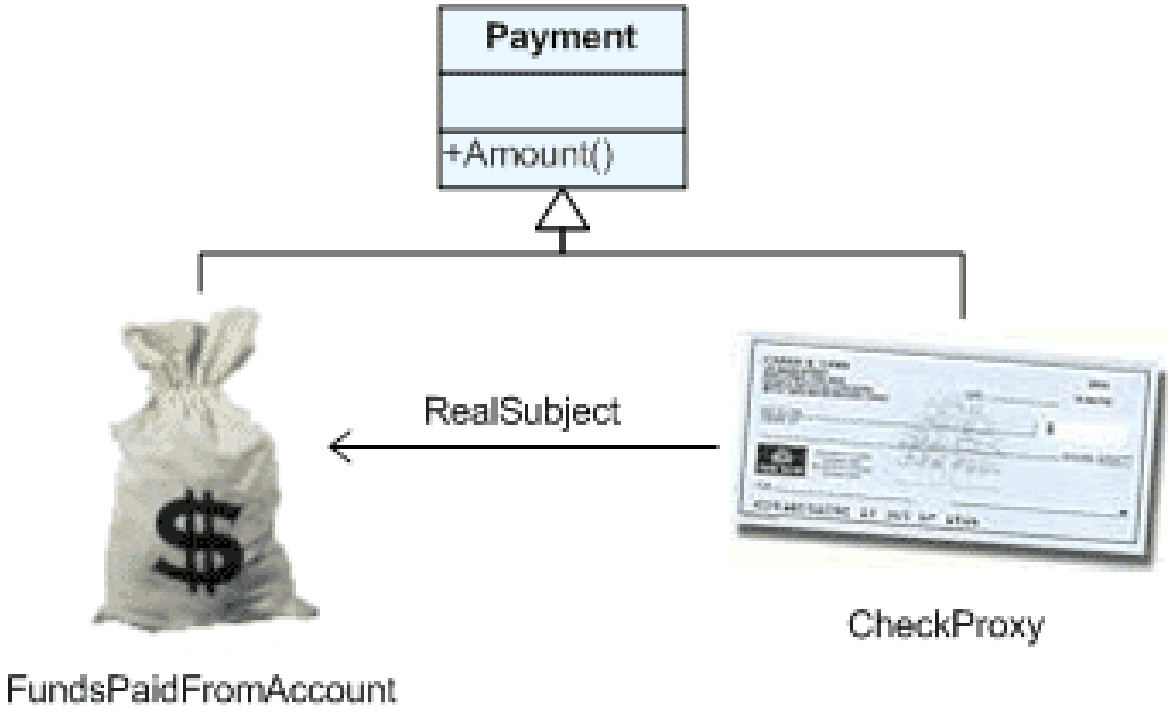
Prototype



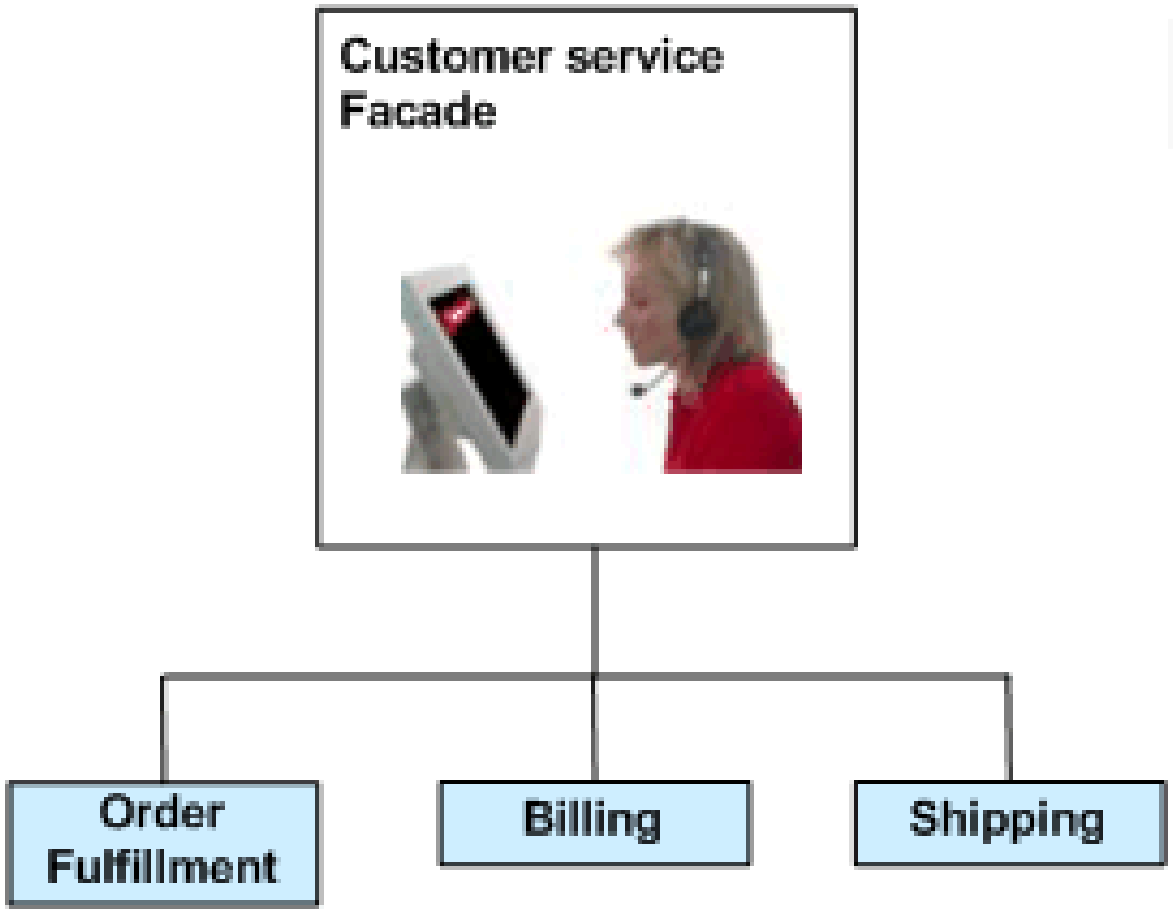


Proxy

Provide a “Placeholder” for an object to control references to it.



Facade


Customer service
Facade
Order
Fulfillment
Billing
Shipping

Selecting suitable design pattern

What Varies	Design Pattern
Algorithms	Strategy, Visitor
Actions	Command
Implementations	Bridge
Response to change	Observer
Interactions between objects	Mediator
Object being created	Factory Method, Abstract Factory, Prototype
Structure being created	Builder
Traversal Algorithm	Iterator
Object interfaces	Adapter
Object behavior	Decorator, State

SYNERGETICS GET IT RIGHT | Microsoft Partner
Gold Cloud Platform
Silver Learning

Thank You