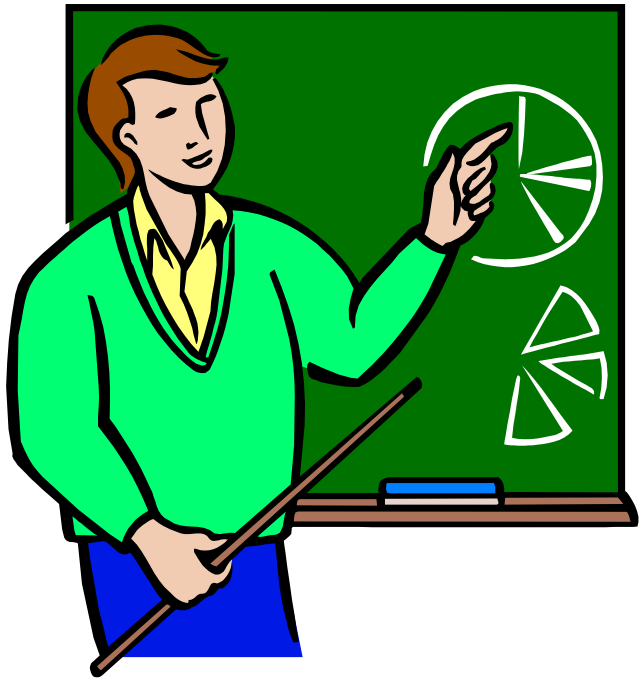
OO Design Principles and Patterns

By Pradeep LN

Primary OO Concepts

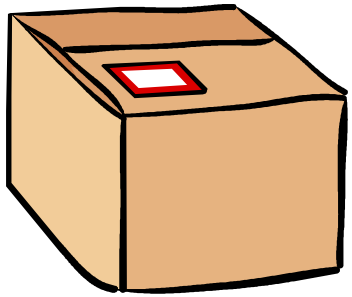
Primary Object Oiented Concepts

- Abstraction
- Encapsulation
- Inheritance
- Cohesion
- Coupling



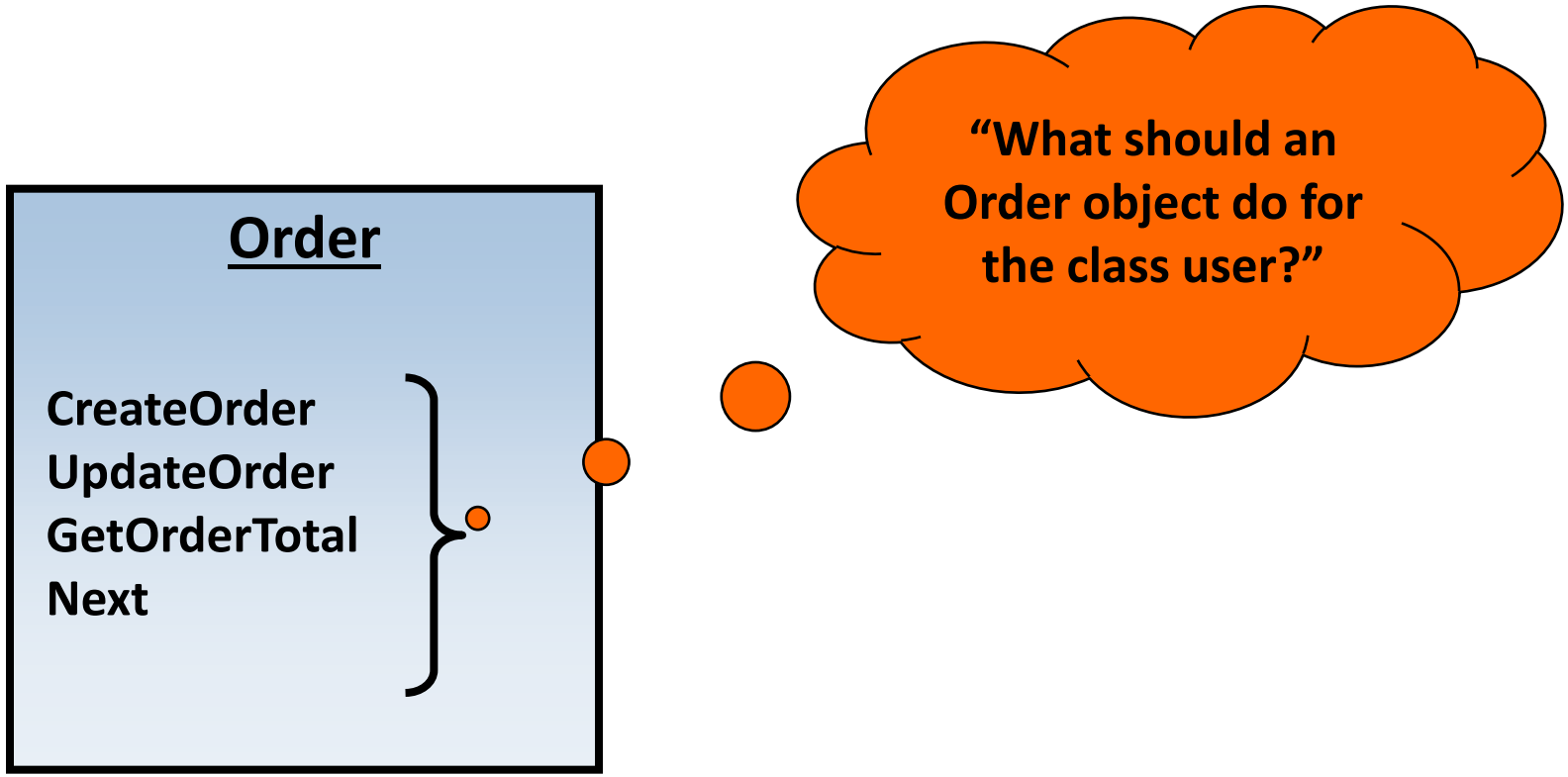
Abstraction

Public View of an Object



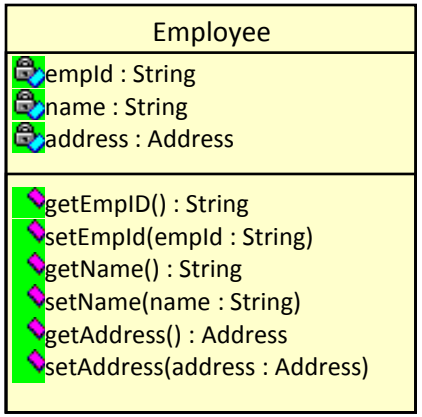
- *Abstraction* is used to minimize complexity for the class user
  - By allowing him focus on the essential characteristics
  - By hiding the details of implementation
- Simply put, abstraction is nothing but a process of ensuring that class users are not exposed to details which they do not need (or use).

Abstraction - Example

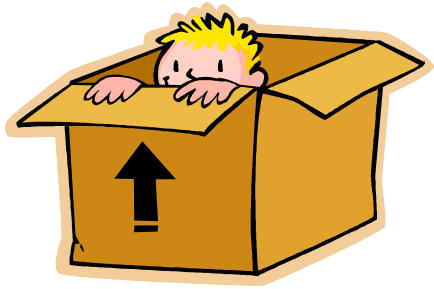


Encapsulation

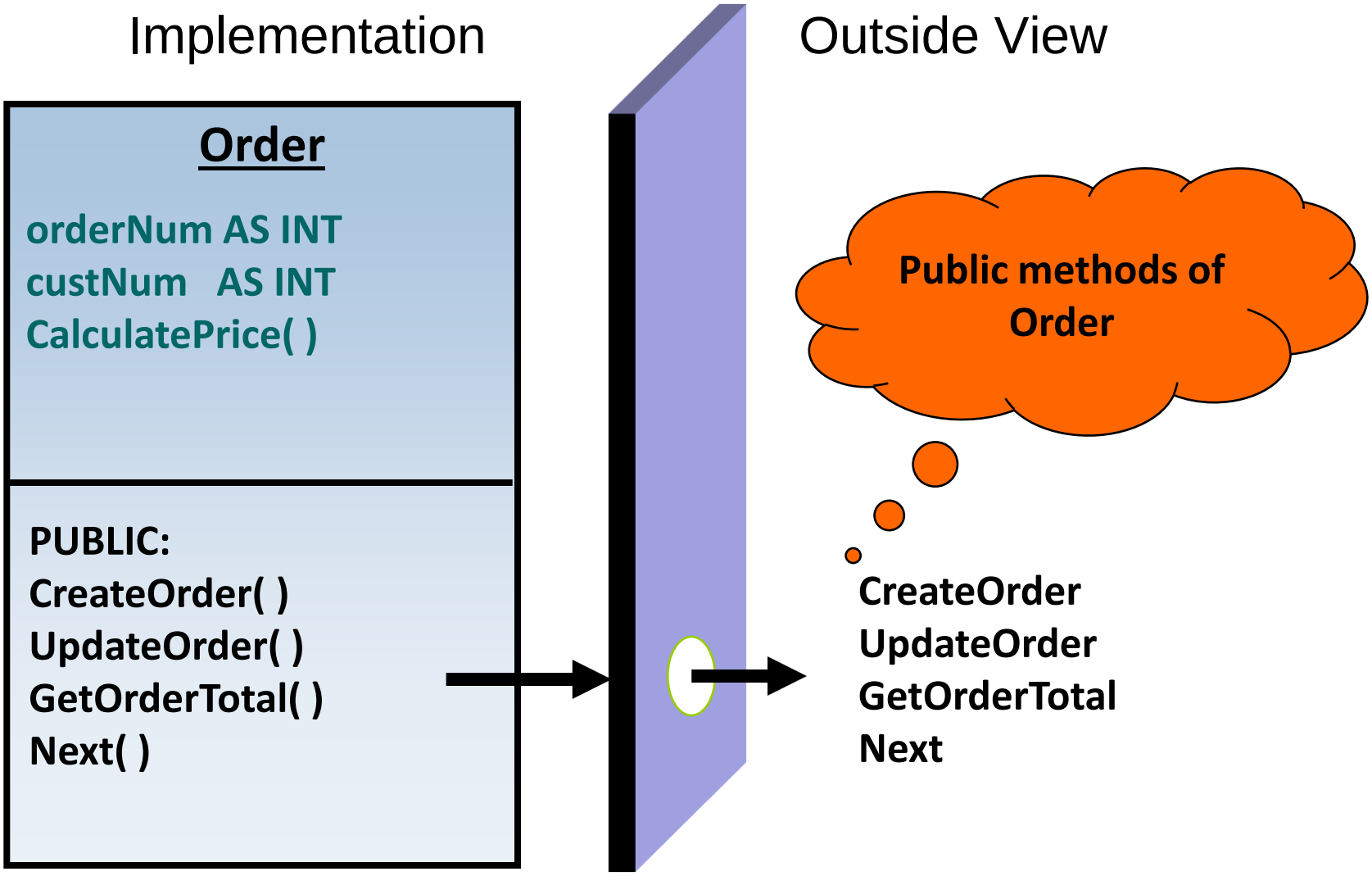
Hide Implementation Details



- Encapsulation is
  - The grouping of related ideas into a single unit, which can thereafter be referred to by a single name.
  - The process of compartmentalizing 'the elements of an abstraction' that constitute its **structure** and **behavior**.
- *Encapsulation* hides implementation
  - Promotes modular software design – data and methods together
  - Data access always done through methods
  - Often called "information hiding"
- Provides two kinds of protection:
  - State cannot be changed directly from outside
  - Implementation can change without affecting users of the object



Encapsulation - Example



Cohesion

- Cohesion is a measure of how strongly-related and focused the responsibilities of a single class are.
- If the methods that serve the given class tend to be similar in many aspects the class is said to have high cohesion. In a highly-cohesive system, code readability and the likelihood of reuse is increased, while complexity is kept manageable.

- Cohesion is decreased if:

- The responsibilities (methods) of a class have little in common.
- Methods carry out many varied activities, often using coarsely-grained or unrelated sets of data.

Cohesion

- Disadvantages of low cohesion (or "weak cohesion") are:
  - Increased difficulty in understanding modules.
  - Increased difficulty in maintaining a system, because logical changes in the domain affect multiple modules, and because changes in one module require changes in related modules.
  - Increased difficulty in reusing a module because most applications won’t need the random set of operations provided by a module.

Coupling

- Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong").
- Low coupling refers to a relationship in which one module interacts with another module through a stable interface and does not need to be concerned with the other module's internal implementation.
- With low coupling, a change in one module will not require a change in the implementation of another module.
- Low coupling is often a sign of a well-structured computer system, and when combined with high cohesion, supports the general goals of high readability and maintainability.

Coupling

- Systems that do not exhibit low coupling might experience the following developmental difficulties:
  - Change in one module forces a ripple of changes in other modules.
  - Modules are difficult to understand in isolation.
  - Modules are difficult to reuse or test because dependent modules must be included.

UML INTRODUCTION

Design Fundamentals

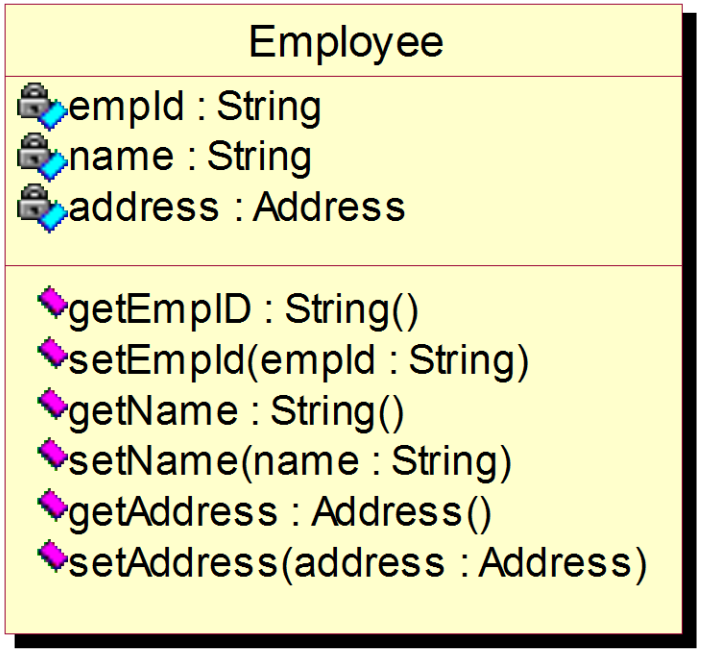
- Topics
  - A quick introduction to UML class diagrams
  - Relationships
    - Generalization
    - Realization
    - Association
    - Aggregation
    - Composition
    - Dependency

Class Diagram

- Every class has three compartments
  - Name of the class
  - Structure (Data members)
  - Behavior (Methods)

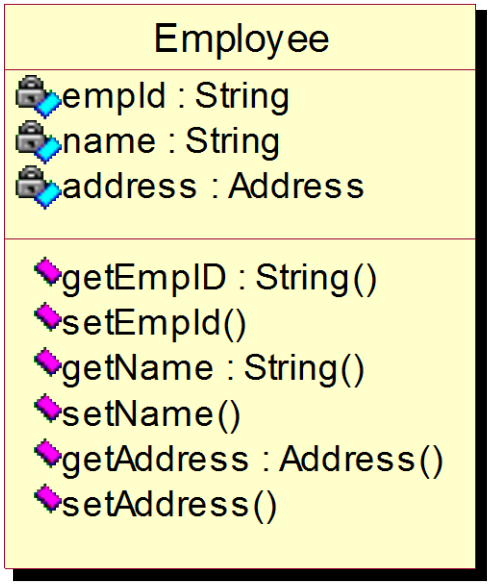
Structure →

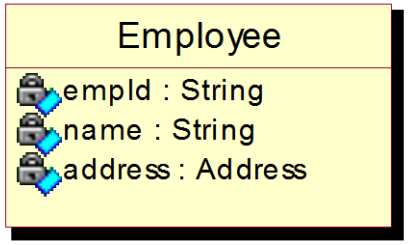
Behavior →

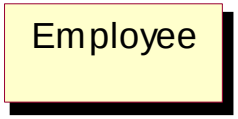


Class Diagram

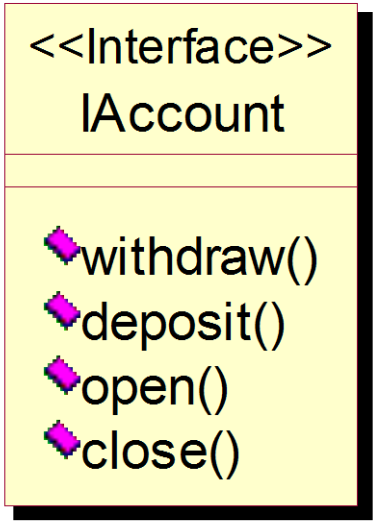
- Other representations

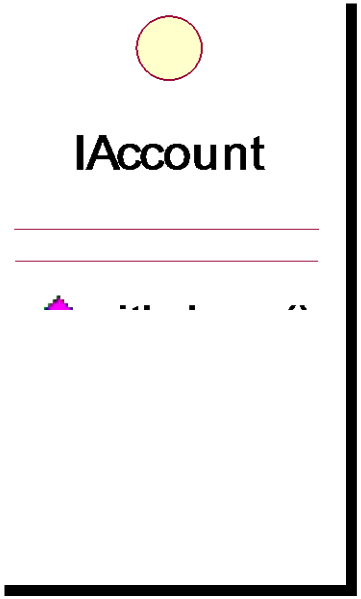




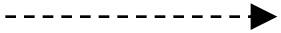
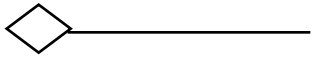
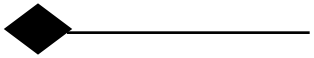
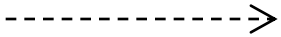


Interface





Relationships

- Six types of relationships in UML
  - Generalization
  - Realization 
  - Association
  - Aggregation 
  - Composite Aggregation or Composition 
  - Dependency 

Relationships

- Classification

<<Is-a>>

Generalization
Realization

<<Has-a>>

Association
Aggregation
Composition

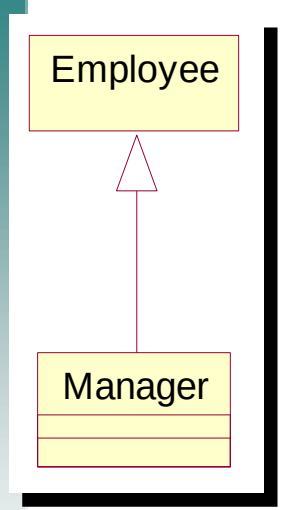
<<Uses>>

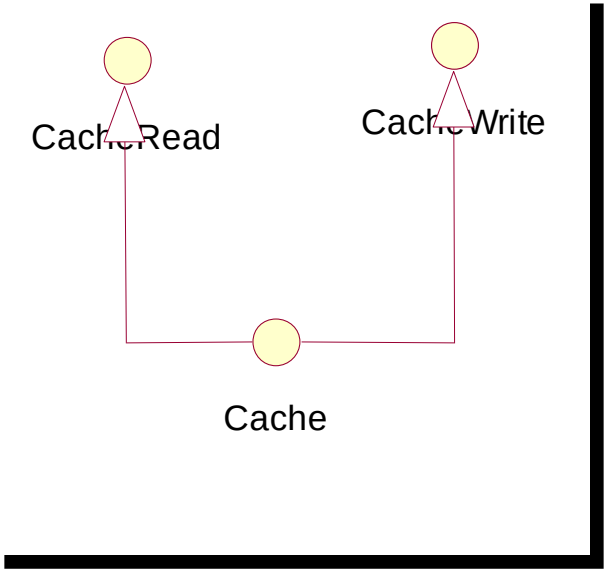
Dependency

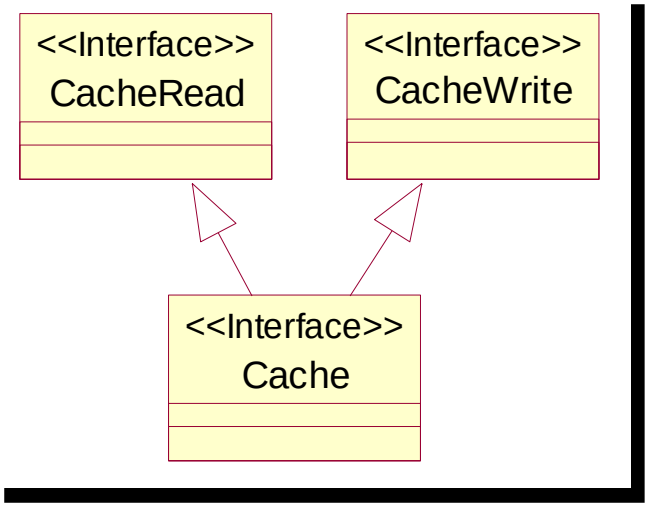
Generalization

Generalization 

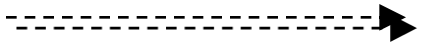
- Relationship between two classes or two interfaces
  - Avoid multiple inheritance between classes
  - Possible to have multiple inheritance between interfaces
  - Interface extends another interface



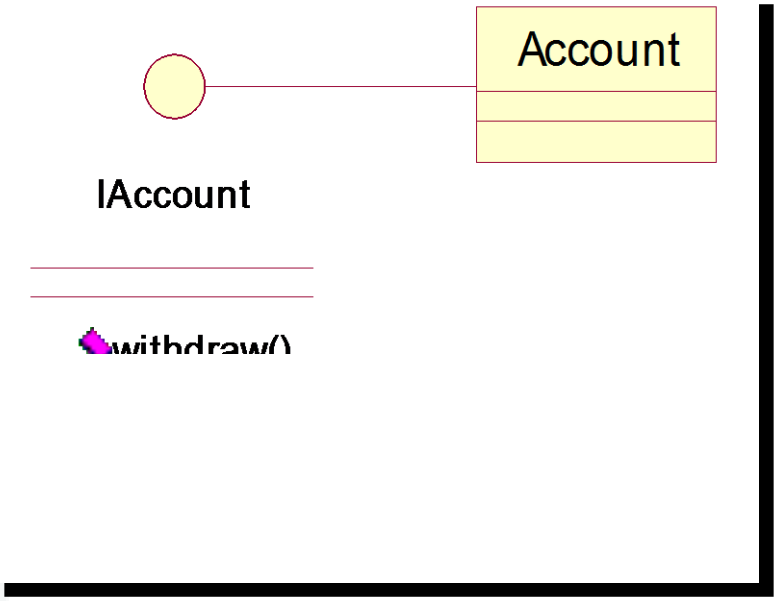


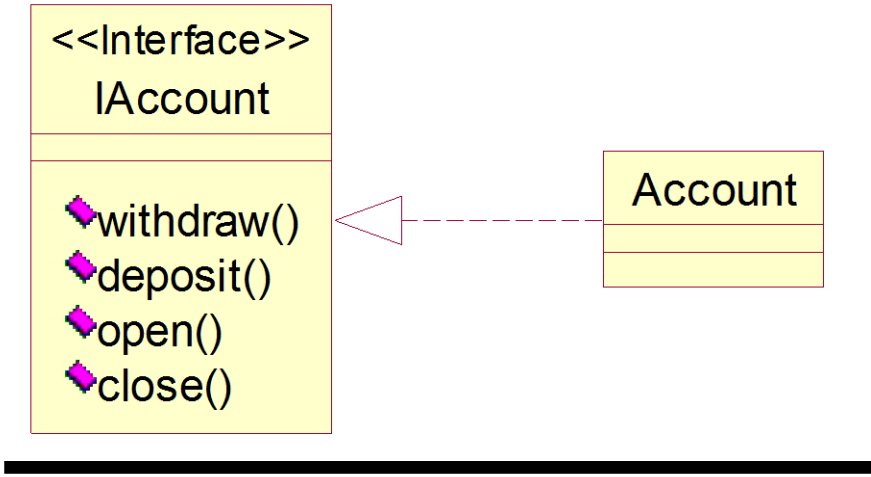


Realization

Realization 

- Relationship between a class and an interface
  - A class can realize multiple interfaces
  - Realizing an interface would require a class to provide an implementation for all the inherited method declarations





Inheritance

- Types of Inheritance
  - Interface Inheritance
    - Realization
  - Implementation Inheritance
    - Generalization

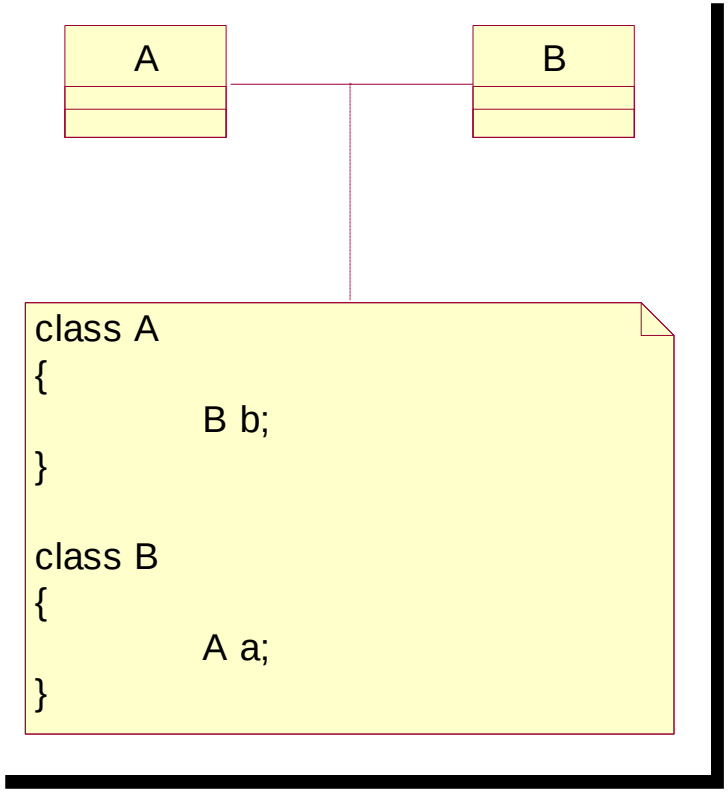
Association

Association ———————————

- ‘Has-a’ relationship
- Semantic relationship between two or more classifiers that involve connections among their instances
- The associations are qualified by
  - Role name
  - Navigability
  - Multiplicity

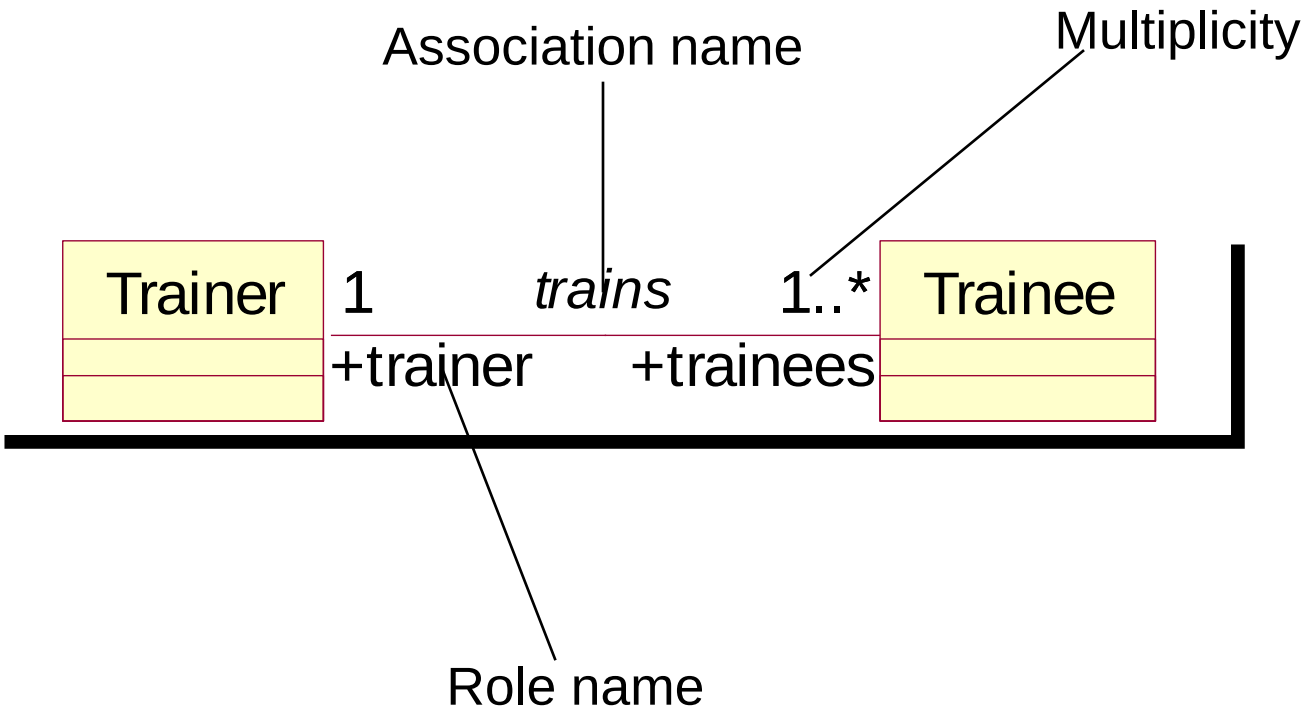
Association

- Multiplicity is by default 1
- Navigability is by default bi-directional



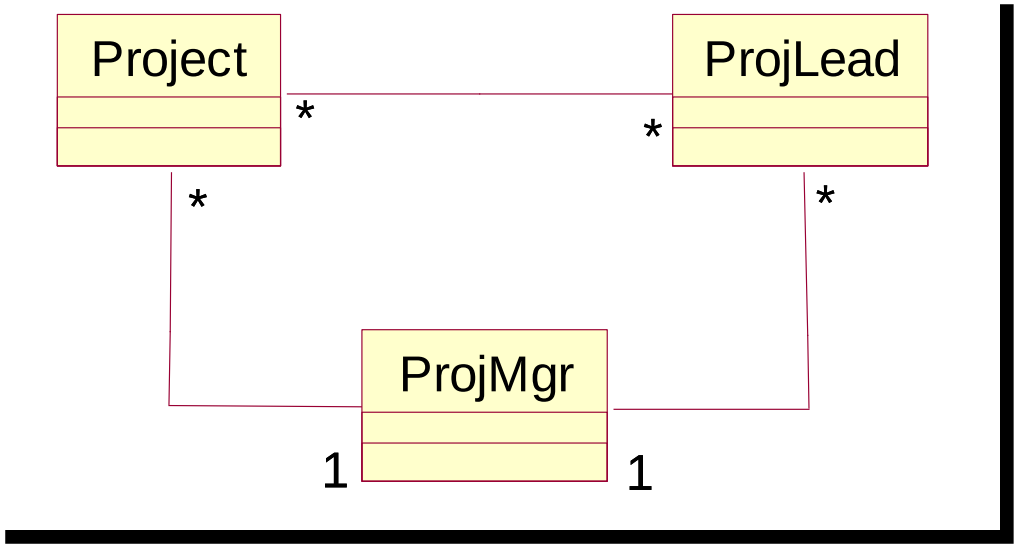
Association

- ‘Has-a’ relationship

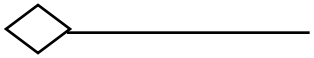


Association

- Examples



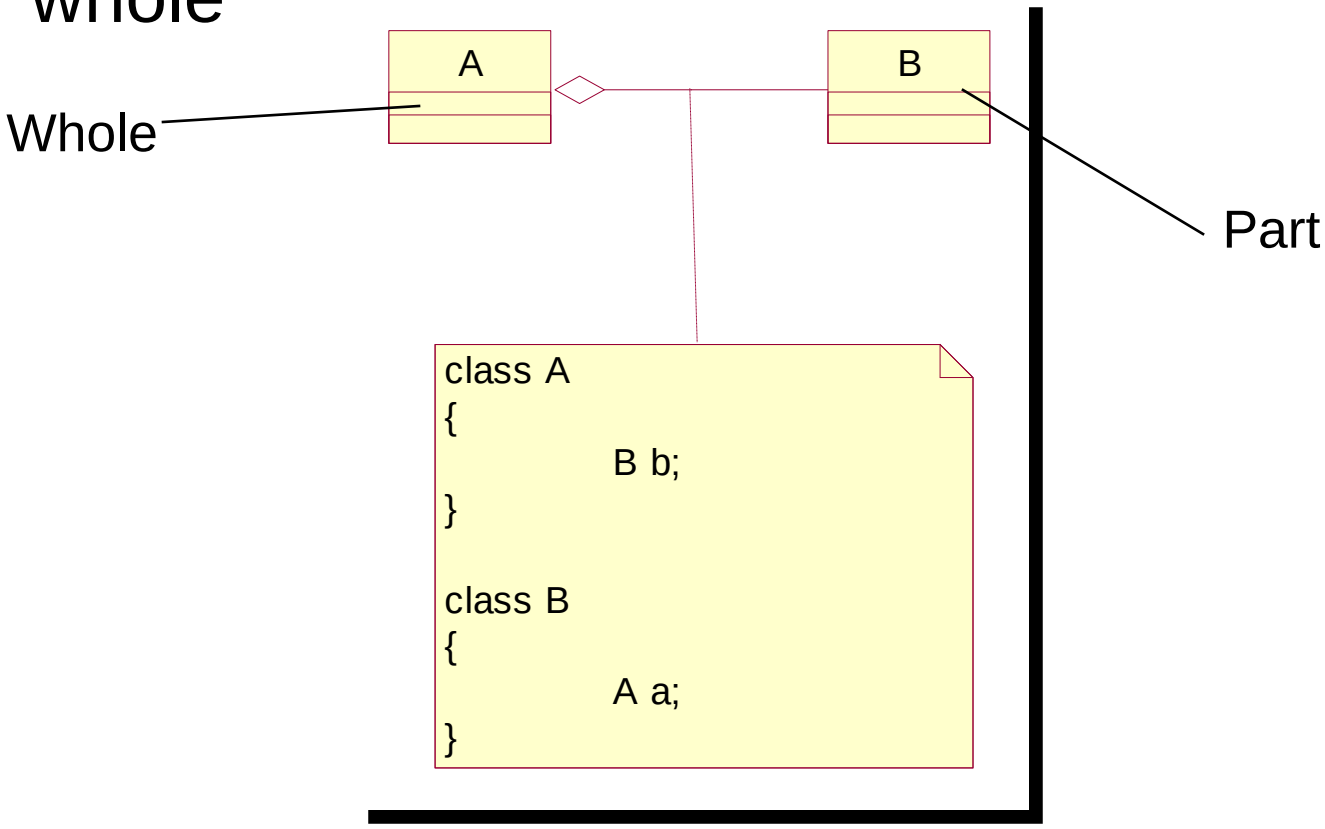
Aggregation

Aggregation 

- ‘Has-a’ relationship
- Is a special (stronger) form of association which conveys a whole part meaning to the relationship
  - Also known as Aggregate Association
- Has multiplicity and navigability
  - Multiplicity is by default 1
  - Navigability is by default bi-directional

Aggregation

- The hollow diamond is placed towards the whole

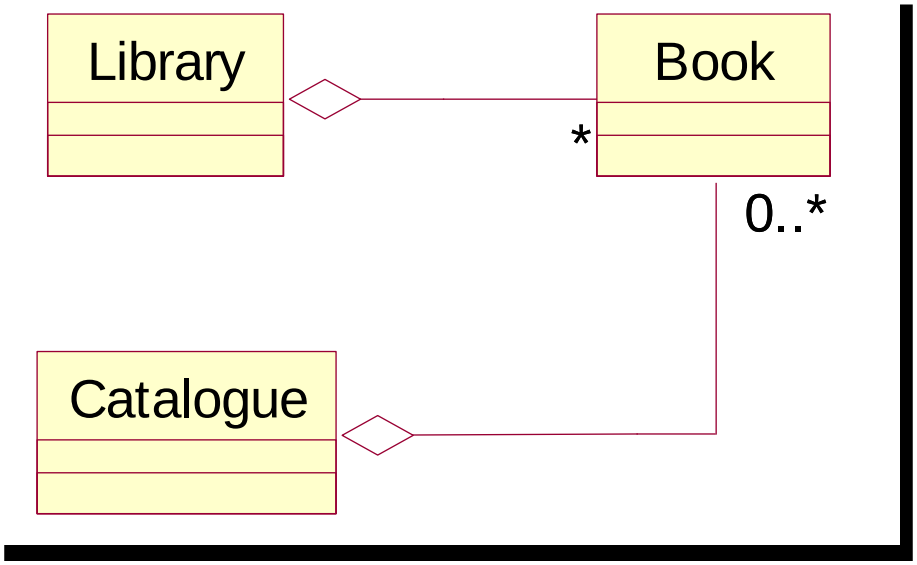


Aggregation

- So what's the difference between Association and Aggregation?
  - When it is comes to code – NOTHING!
- Based on the context
  - Whole Part
  - Lack of independent use and existence
  - Scope of verb is constrained to only 'has' or 'contains'
- When in doubt, leave aggregation out!

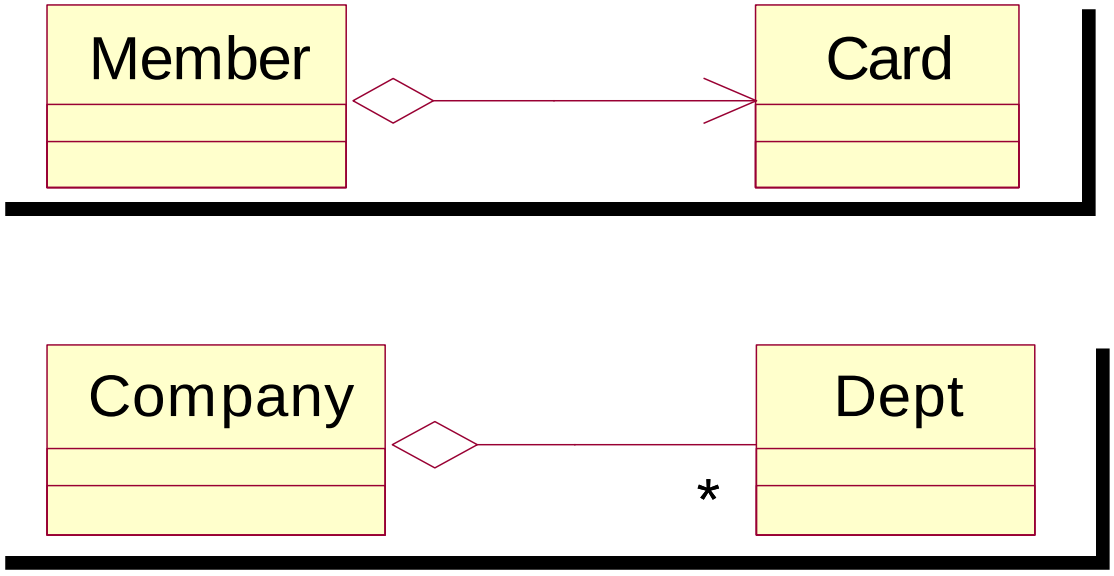
Aggregation

- The Whole – Part nature



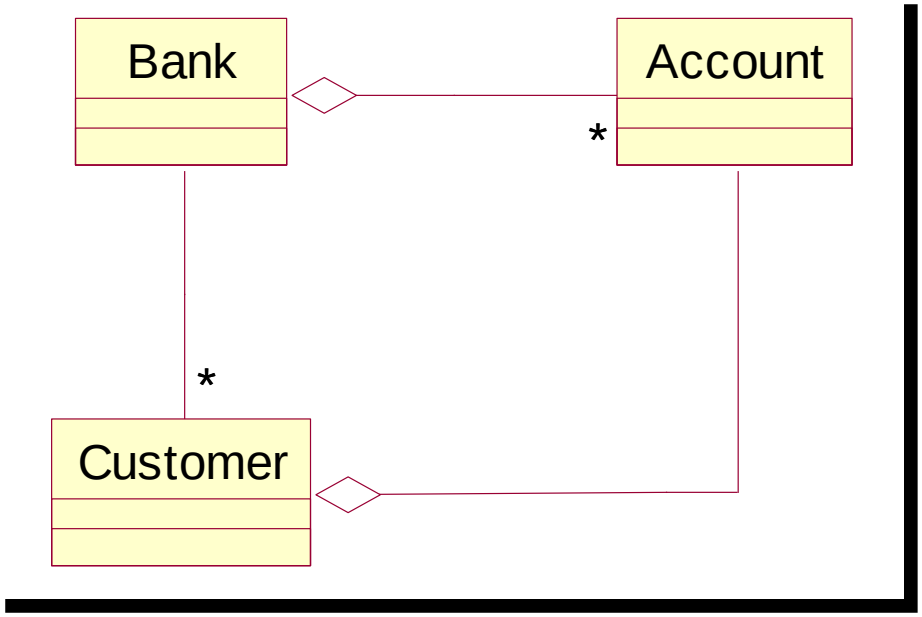
Aggregation

- Independent existence / use

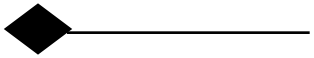


Aggregation

- A part can be shared between many wholes
  - Shared aggregation



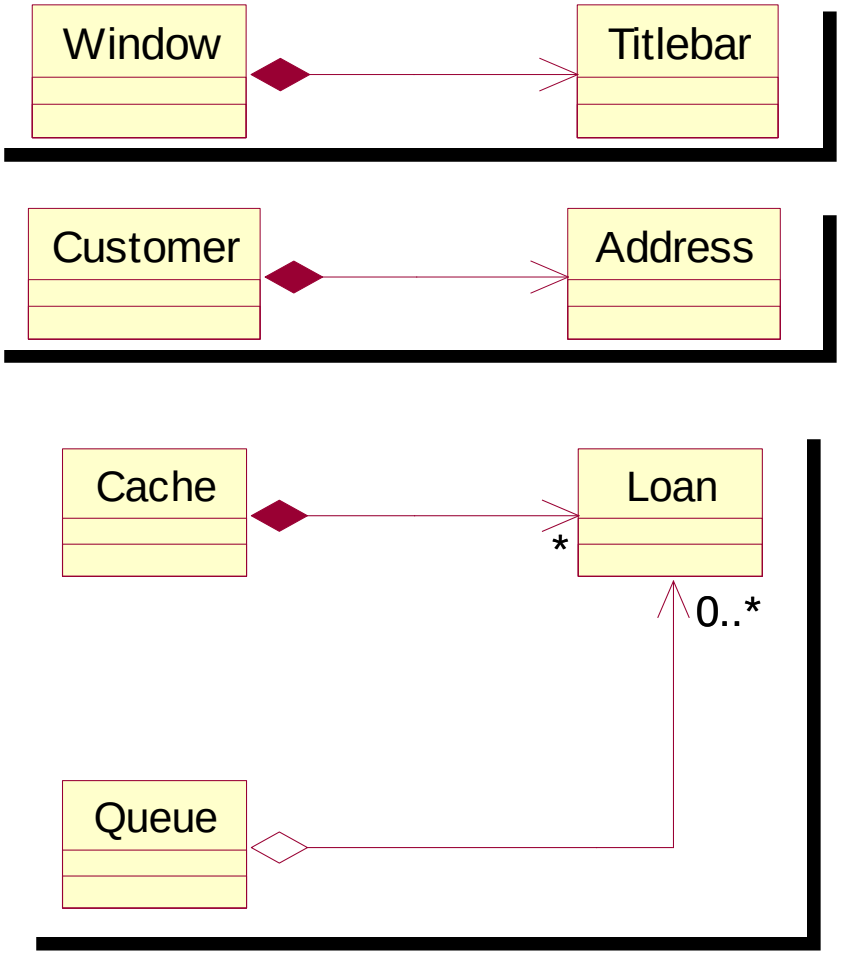
Composite Aggregation

Composite Aggregation 

- ‘Has-a’ relationship
- Is a stronger form of aggregation
  - Also known as Composition
- Explicit lifetime control
  - Part is created when whole is created
  - Part is destroyed when whole is destroyed
  - Exclusive ownership
- Has multiplicity and navigability
  - Multiplicity at the whole end should always be 1
  - Navigability is by default bi-directional

Composite Aggregation

- Examples



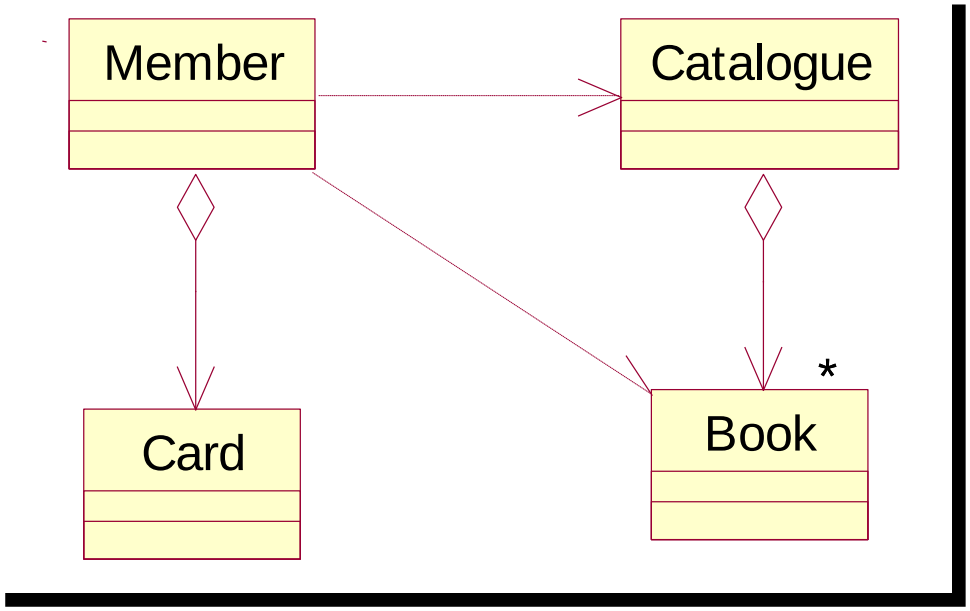
Dependency

Dependency ---------->

- ‘Uses’ relationship
- Behavioral relationship
  - Loose coupling
  - Has no impact on class structure (data members)
- A class references another class only within its methods for the purpose of:
  - Invoking a static method
  - Local instantiation
  - Formal argument use
  - Return type

Dependency

- ‘Uses’ relationship



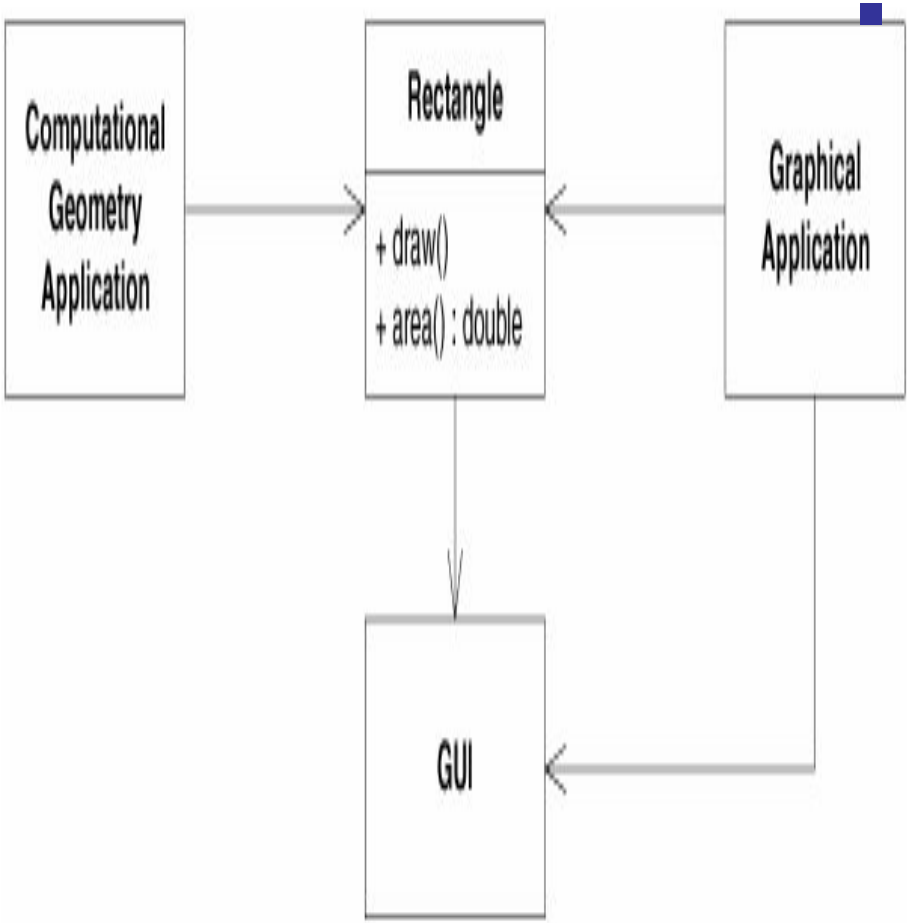
SOLID Design Principles

Single Responsibility Principle (SRP)

The Single-Responsibility Principle

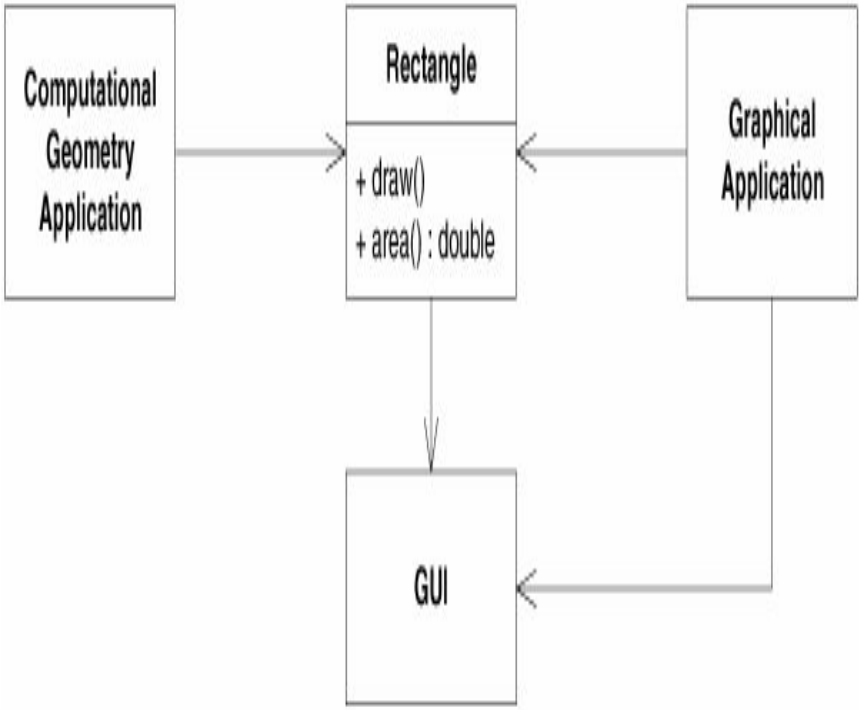
- Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class.
  - All its services should be narrowly aligned with that responsibility
- *"A class should have only one reason to change"*
  - The reason is that each responsibility is an axis of change
  - If a class has more than one responsibility, the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class's ability to meet the others

SRP - Example



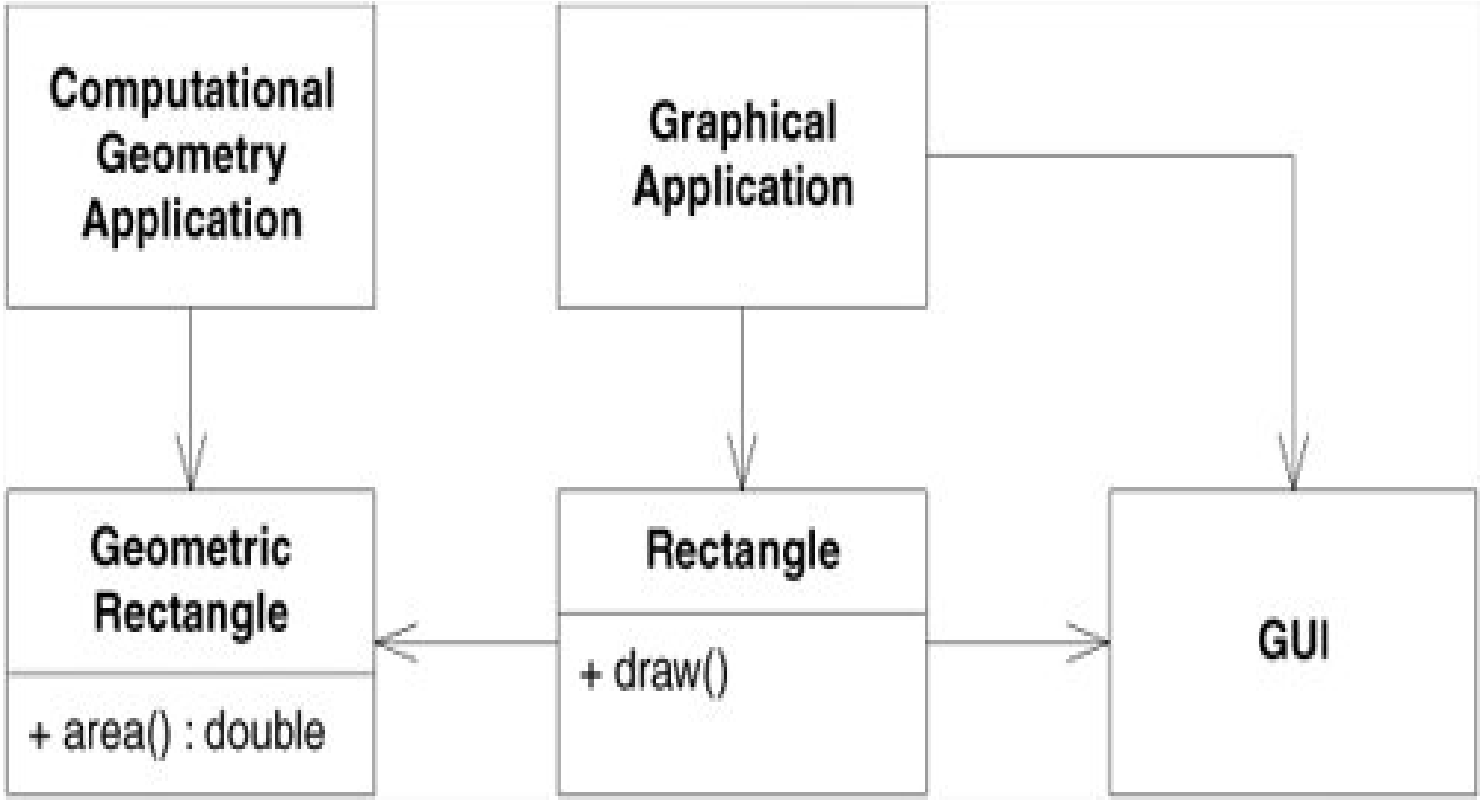
- This design violates SRP. The Rectangle class has two responsibilities.
  - The first responsibility is to provide a mathematical model of the geometry of a rectangle.
  - The second responsibility is to render the rectangle on a GUI.

SRP - Example



- The violation of SRP causes several nasty problems.
- First, we must include GUI in the computational geometry application
- Second, if a change to the GraphicalApplication causes the Rectangle to change for some reason, that change may force us to rebuild, retest, and redeploy the ComputationalGeometryApplication.
  - If we forget to do this, that application may break in unpredictable ways

Separated responsibilities



Defining a Responsibility

- In the context of the SRP, we define a responsibility to be a reason for change.
- If you can think of more than one motive for changing a class, that class has more than one responsibility.
- This is sometimes difficult to see.
  - We are accustomed to thinking of responsibility in groups

Example - 2

```
public interface Modem
{
  public void Dial(string pno);
  public void Hangup();
  public void Send(char c);
  public char Recv();
}
```

- The four functions it declares are certainly functions belonging to a modem
- However, two responsibilities are being shown here.
- The first responsibility is connection management.
- The second is data communication.
- The dial and hangup functions manage the connection of the modem; the send and recv functions communicate data

Example - 2

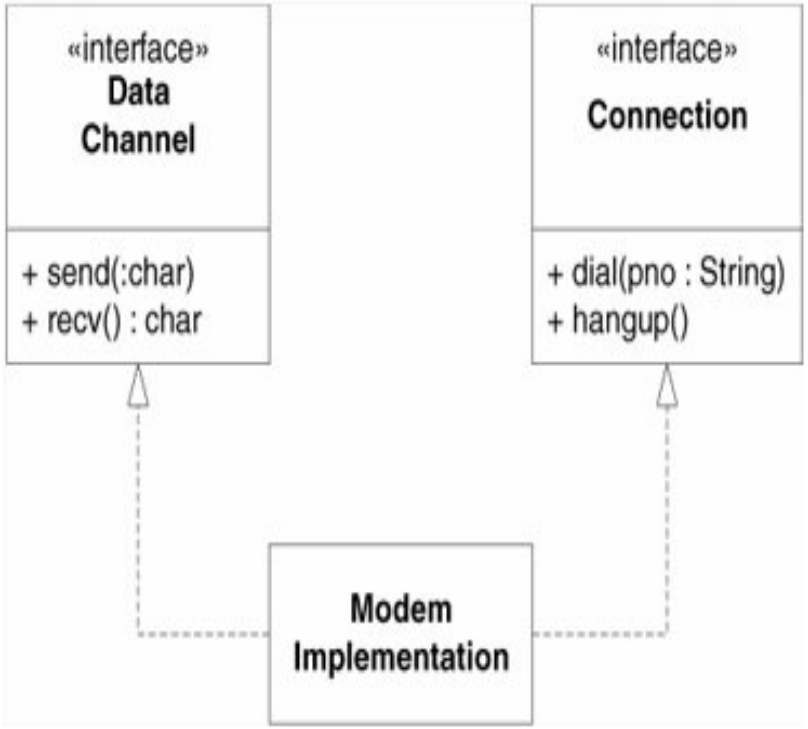
- Should these two responsibilities be separated?
- That depends on how the application is changing.
- If the application changes in ways that affect the signature of the connection functions, the classes that call send and read will have to be recompiled and redeployed more often than we like.
- In that case, the two responsibilities should be separated
- If, on the other hand, the application is not changing in ways that cause the two responsibilities to change at different times, there is no need to separate them.

Example - 2

An axis of change is an axis of change only if the changes occur

- It is not wise to apply SRP or any other principle, for that matter if there is no symptom

Example - 2



- Note that, I kept both responsibilities coupled in the ModemImplementation class.
- This is not desirable, but it may be necessary.
- There are often reasons, having to do with the details of the hardware or operating system, that force us to couple things that we'd rather not couple.
- However, by separating their interfaces, we have decoupled the concepts as far as the rest of the application is concerned.
- however, note that all dependencies flow away from it. Nobody needs to depend on this class

Open/Closed Principle (OCP)

OCP - motivations

- How many times do you start writing a brand new application from nothing versus the number of times you start by adding new functionality to an existing codebase?
- Chances are good that you spend far more time adding new features to an existing codebase.

OCP - motivations

- Is it easier to write all new code or to make changes to existing code?
- It's usually far easier for me to write all new methods and classes than it is to break into old code and find the sections I need to change.
  - Modifying old code adds the risk of breaking existing functionality.
  - With new code you generally only have to test the new functionality.
  - When you modify old code you have to both test your changes and then perform a set of regression tests to make sure you didn't break any of the existing code.

The Open/Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Description of OCP

- Modules that conform to OCP have two primary attributes.
  - They are open for extension. This means that the behavior of the module can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does.
  - They are closed for modification. Extending the behavior of a module does not result in changes to the source, or binary, code of the module.

Liskov Substitution Principle (LSP)

Liskov Substitution Principle

- The primary mechanisms behind the Open/Closed Principle are abstraction and polymorphism
- key mechanisms that supports abstraction and polymorphism is inheritance
- What are the design rules that govern this particular use of inheritance?
- What are the characteristics of the best inheritance hierarchies?
- What are the traps that will cause us to create hierarchies that do not conform to OCP? These are the questions addressed by the Liskov Substitution Principle (LSP).

The Liskov Substitution Principle

- ***Subtypes must be substitutable for their base types.***
- Barbara Liskov wrote this principle in 1988. She said:
  - What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.

The Interface Segregation Principle (ISP)

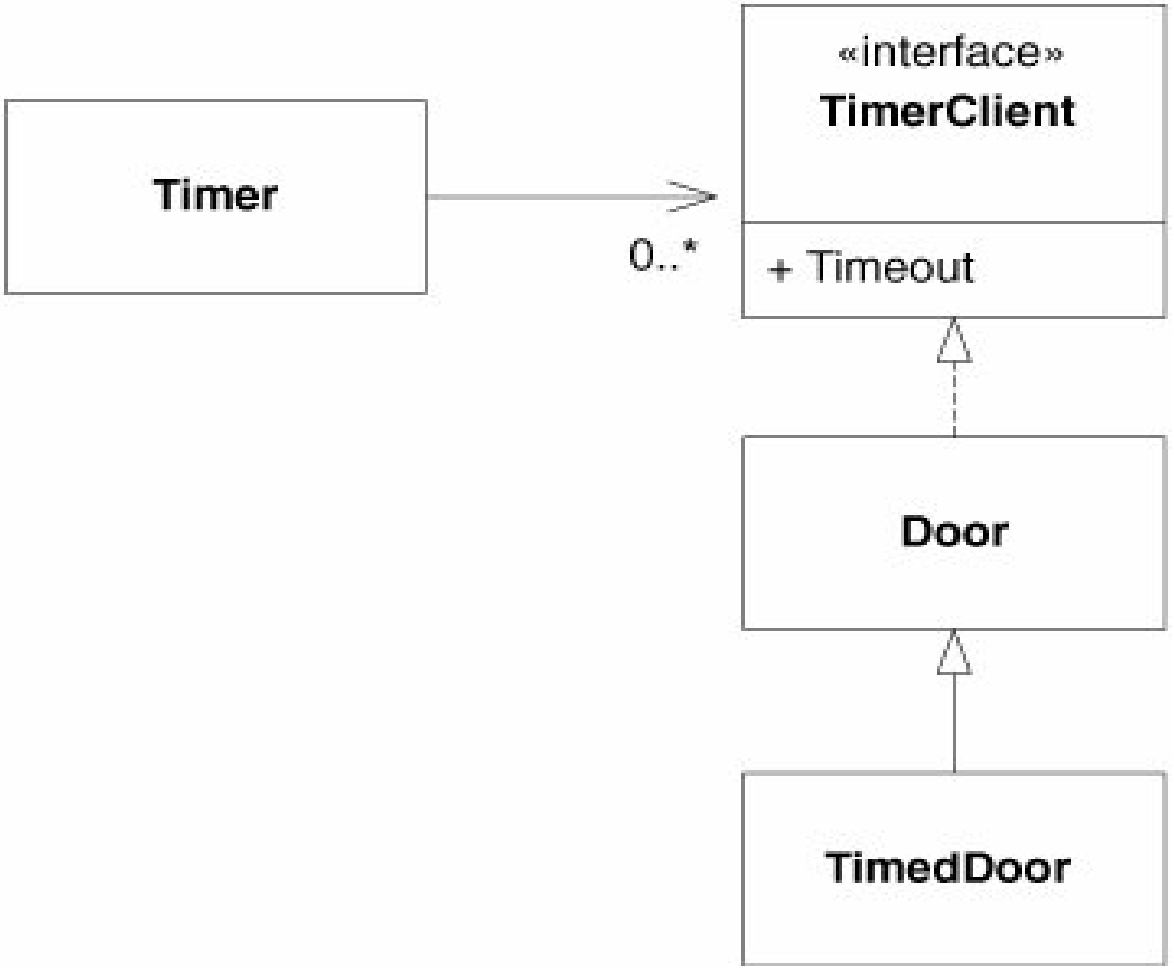
The Interface Segregation Principle (ISP)

- This principle deals with the disadvantages of "fat" interfaces.
- Classes whose interfaces are not cohesive have "fat" interfaces.
  - In other words, the interfaces of the class can be broken up into groups of methods. Each group serves a different set of clients.
  - ISP acknowledges that there are objects that require noncohesive interfaces; however, it suggests that clients should not know about them as a single class.

Interface Pollution

- Consider a security system in which Door objects can be locked and unlocked and know whether they are open or closed.
- This Door is coded as an interface so that clients can use objects that conform to the Door interface without having to depend on particular implementations of Door.
- Now consider that one such implementation, TimedDoor, needs to sound an alarm when the door has been left open for too long. In order to do this, the TimedDoor object communicates with another object called a Timer.

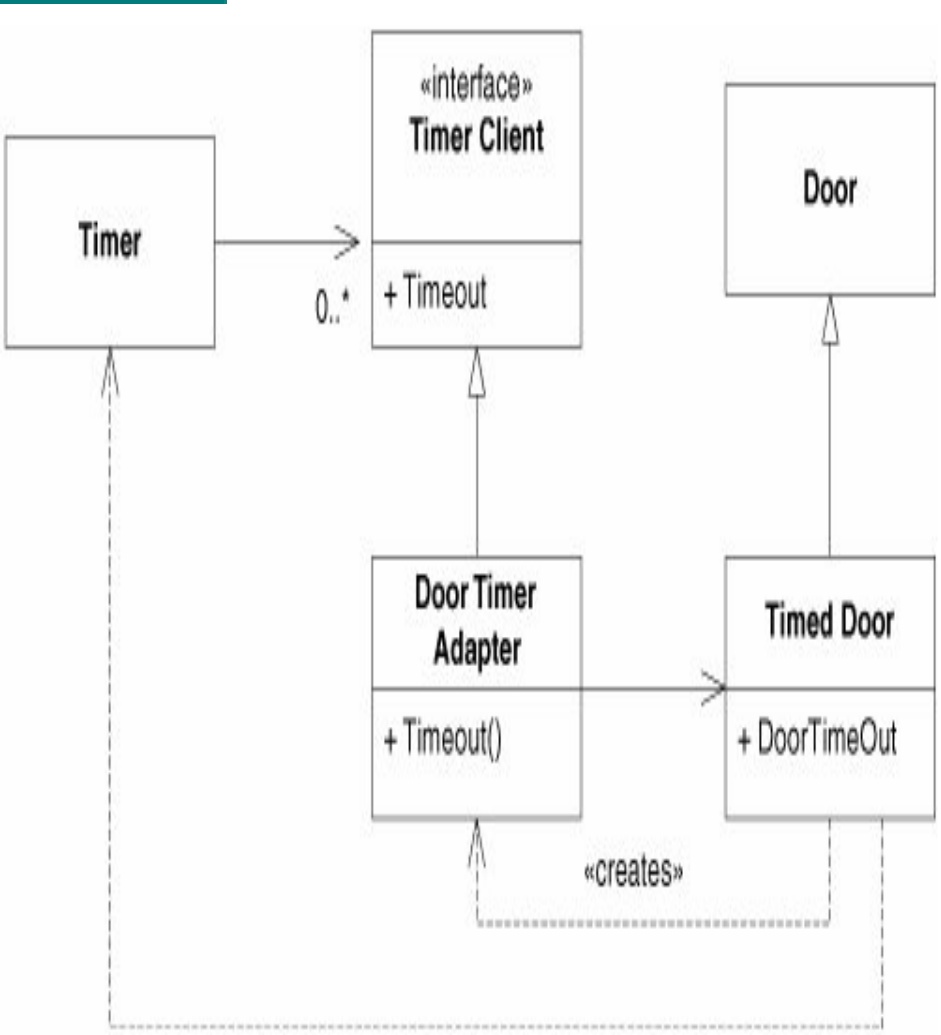
Interface Pollution

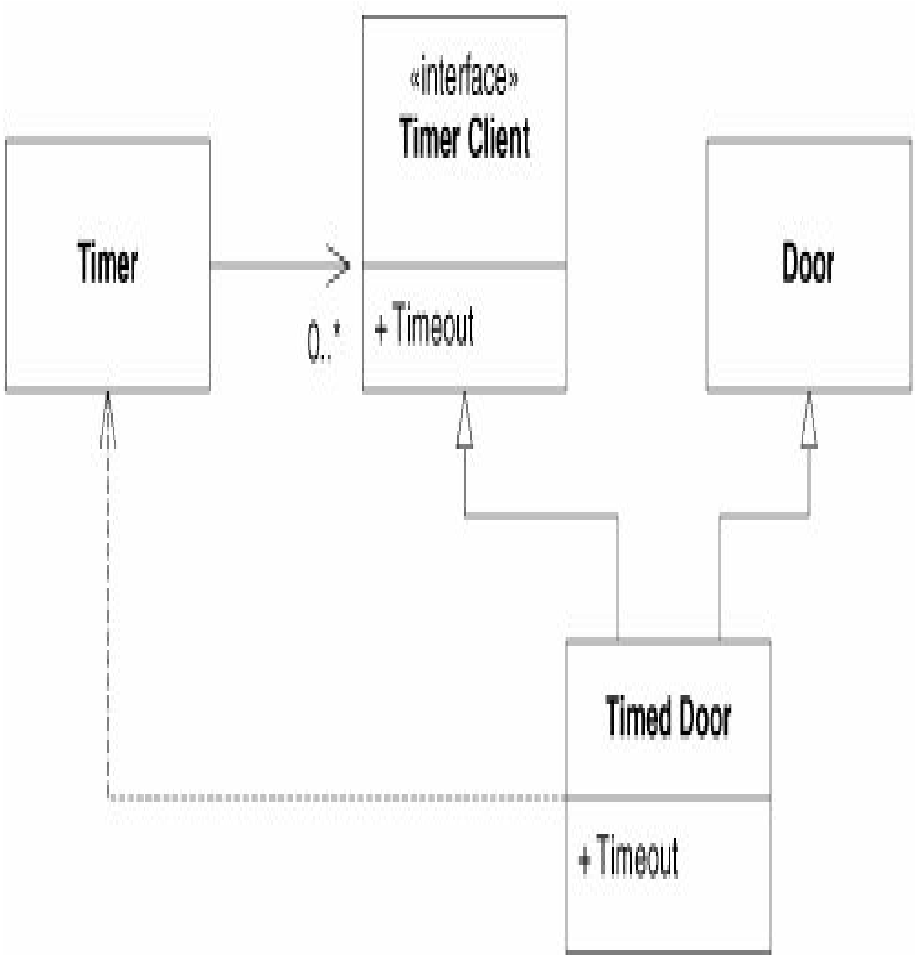


The Interface Segregation Principle

- ***Clients should not be forced to depend on methods they do not use.***
- When clients are forced to depend on methods they don't use, those clients are subject to changes to those methods.
  - This results in an inadvertent coupling between all the clients.
  - Said another way, when a client depends on a class that contains methods that the client does not use but that other clients do use, that client will be affected by the changes that those other clients force on the class.
- We would like to avoid such couplings where possible, and so we want to separate the interfaces.

2 Possible solutions



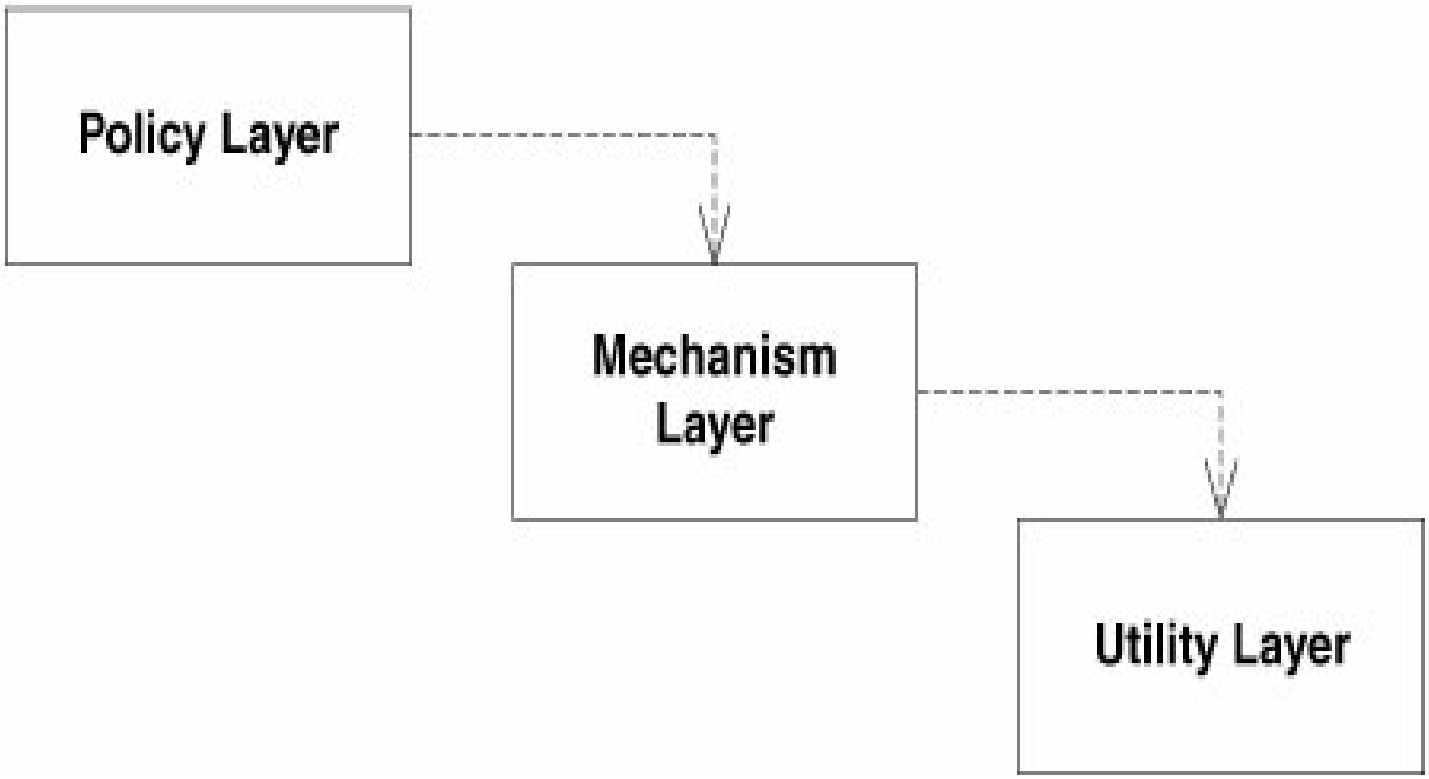


The Dependency-Inversion Principle (DIP)

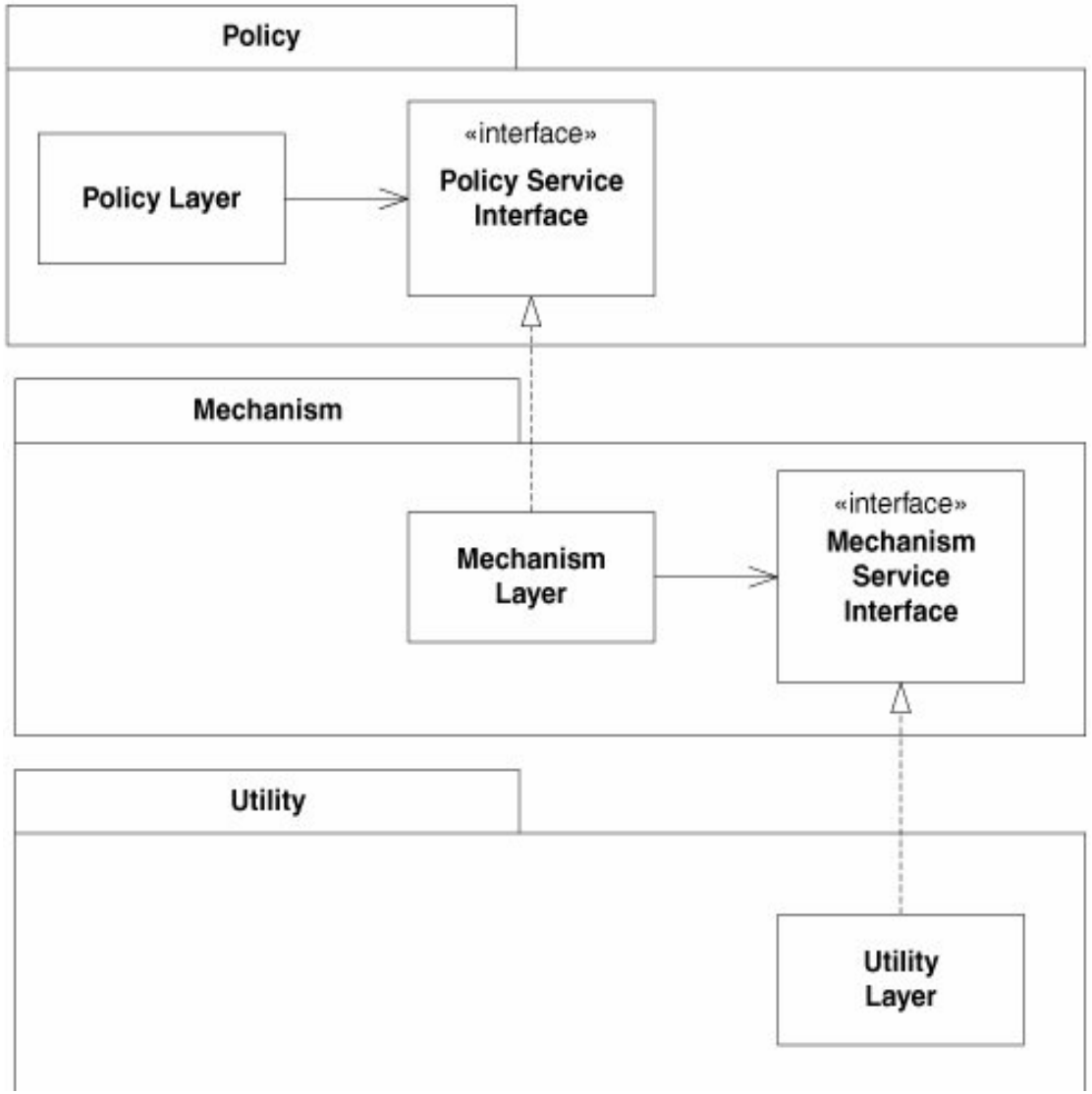
The Dependency-Inversion Principle

- **The Dependency-Inversion Principle**
  - ***High-level modules should not depend on low-level modules. Both should depend on abstractions.***
  - ***Abstractions should not depend upon details. Details should depend upon abstractions.***

In conventional architecture, higher-level components depend upon lower-level components as depicted in the following diagram:



Inverted layers



Ownership Inversion

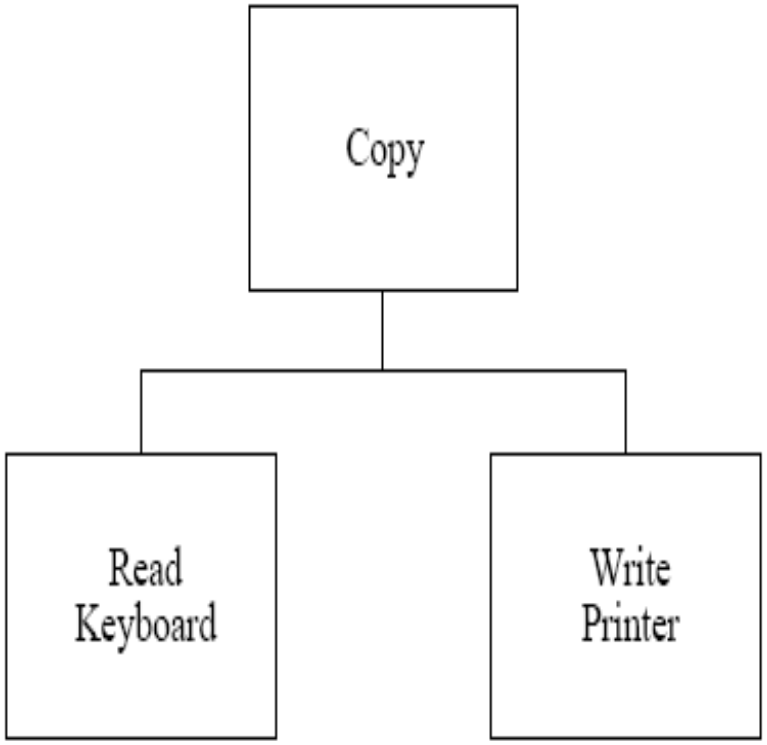
- Note that the inversion here is one of not only dependencies but also interface ownership. We often think of utility libraries as owning their own interfaces.
- But when DIP is applied, we find that the clients tend to own the abstract interfaces and that their servers derive from them.
- This is sometimes known as the Hollywood principle: "Don't call us; we'll call you.“
- The lower-level modules provide the implementation for interfaces that are declared within, and called by, the upper-level modules.

Dependence on Abstractions

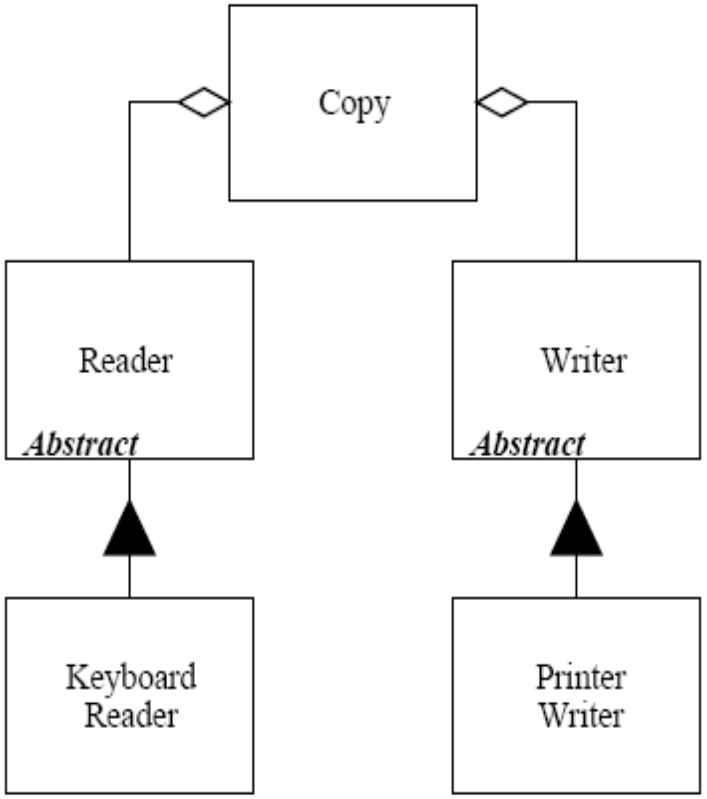
- A somewhat more naive, yet still very powerful, interpretation of DIP is the simple heuristic: "Depend on abstractions." Simply stated, this heuristic recommends that you should not depend on a concrete class and that rather, all relationships in a program should terminate on an abstract class or an interface.
  - No variable should hold a reference to a concrete class.
  - No class should derive from a concrete class.
  - No method should override an implemented method of any of its base classes.
- Certainly, this heuristic is usually violated at least once in every program. Somebody has to create the instances of the concrete classes, and whatever module does that will depend on them

An Example

Before



After



Design Patterns

Why Design Patterns?

- A good scientist is a person with original ideas.
- A good engineer is a person who makes a design that works with as few original ideas as possible.

—*Freeman Dyson*

What Is a Design Pattern?

- Generalized solution to a recurring problem in a particular context
- Product of community experience
- Four basic elements
  - *name:* a “handle” for the pattern
  - *problem:* the problem and its context
  - *solution*: elements, responsibilities, and collaborations
  - *consequences*: benefits and tradeoffs

GoF Patterns

- Often referred to as the GoF, or Gang-Of-Four because of the four authors who wrote the book, Design Patterns: Elements of Reusable Object-Oriented Software
- The book's authors are Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Creational patterns

- These patterns have to do with class instantiation.
- Examples:
  - Builder constructs complex objects by separating construction and representation.
  - Factory Method creates objects without specifying the exact class to create.
  - Singleton restricts object creation for a class to only one instance.

Structural patterns

- These concern Class and Object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.
- Examples:
  - Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
  - Bridge decouples an abstraction from its implementation so that the two can vary independently.
  - Decorator dynamically adds/overrides behaviour in an existing method of an object.
  - Flyweight reduces the cost of creating and manipulating a large number of similar objects.
  - Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Behavioral patterns

- These design patterns are about Class's objects communication. They are specifically concerned with communication between objects.
- Examples:
  - Command creates objects which encapsulate actions and parameters.
  - Iterator accesses the elements of an object sequentially without exposing its underlying representation.

GoF Patterns

Bridge

- **Intent**
- Decouple an abstraction from its implementation so that the two can vary independently. [GoF, p151]
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
- Beyond encapsulation, to insulation
- **Problem**
- "Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

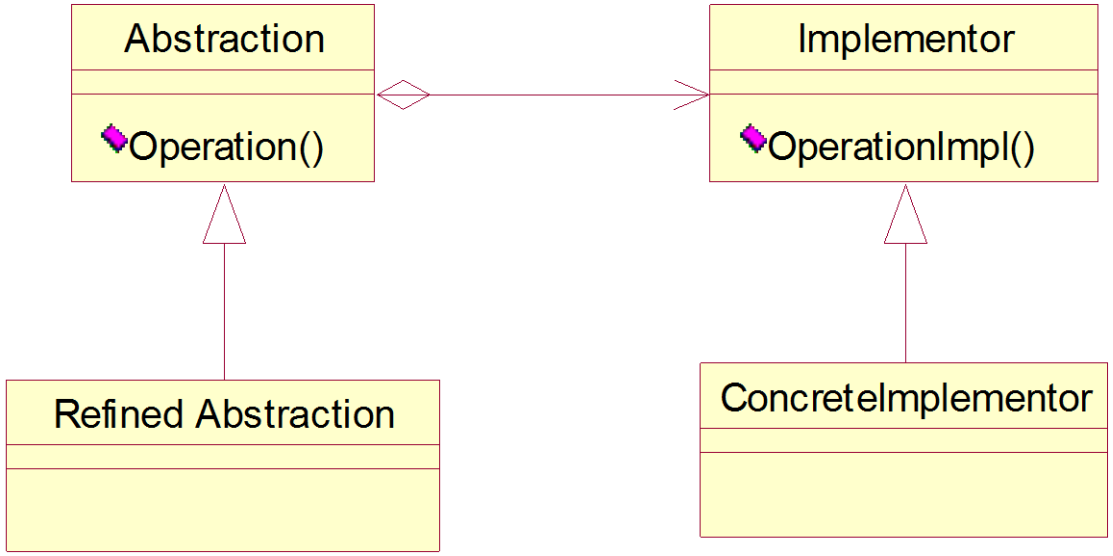
Applicability

This pattern is used when

- You want to avoid permanent binding between abstraction and implementation
- Changes in the implementation of an abstraction must have no effect on the client
- Implementation has to be shared across multiple objects

Structural Patterns – Bridge

Structure



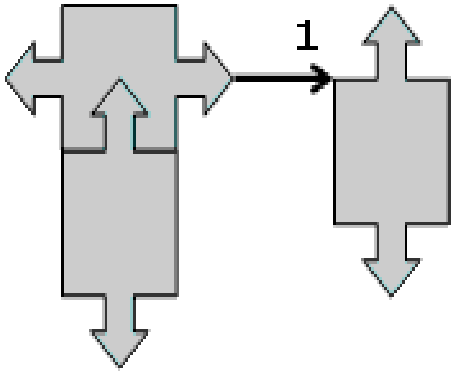
Bridge

1) Divide a component into abstraction and implementation.

2) Massage all the possible implementations into an inheritance hierarchy: put each possible implementation in a derived class, and define an interface in an abstract base class that makes instances of the derived classes interchangeable.

3) Define an inheritance hierarchy for the component's abstraction: specializations in derived classes, and generalization in a base class.

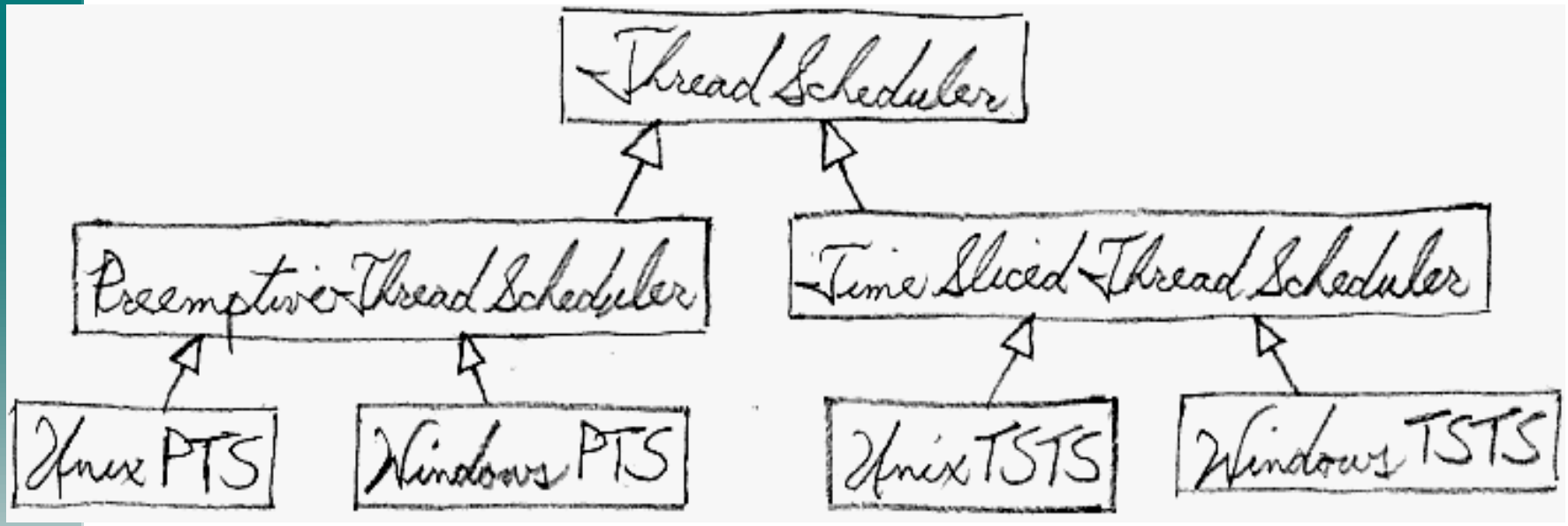
4) Position the abstraction hierarchy as a wrapper to the implementation hierarchy.



5) The client will choose the wrapper object, and may configure or modify the wrappee object.

6) As the client makes requests, the abstraction object simply delegates to the implementation object.

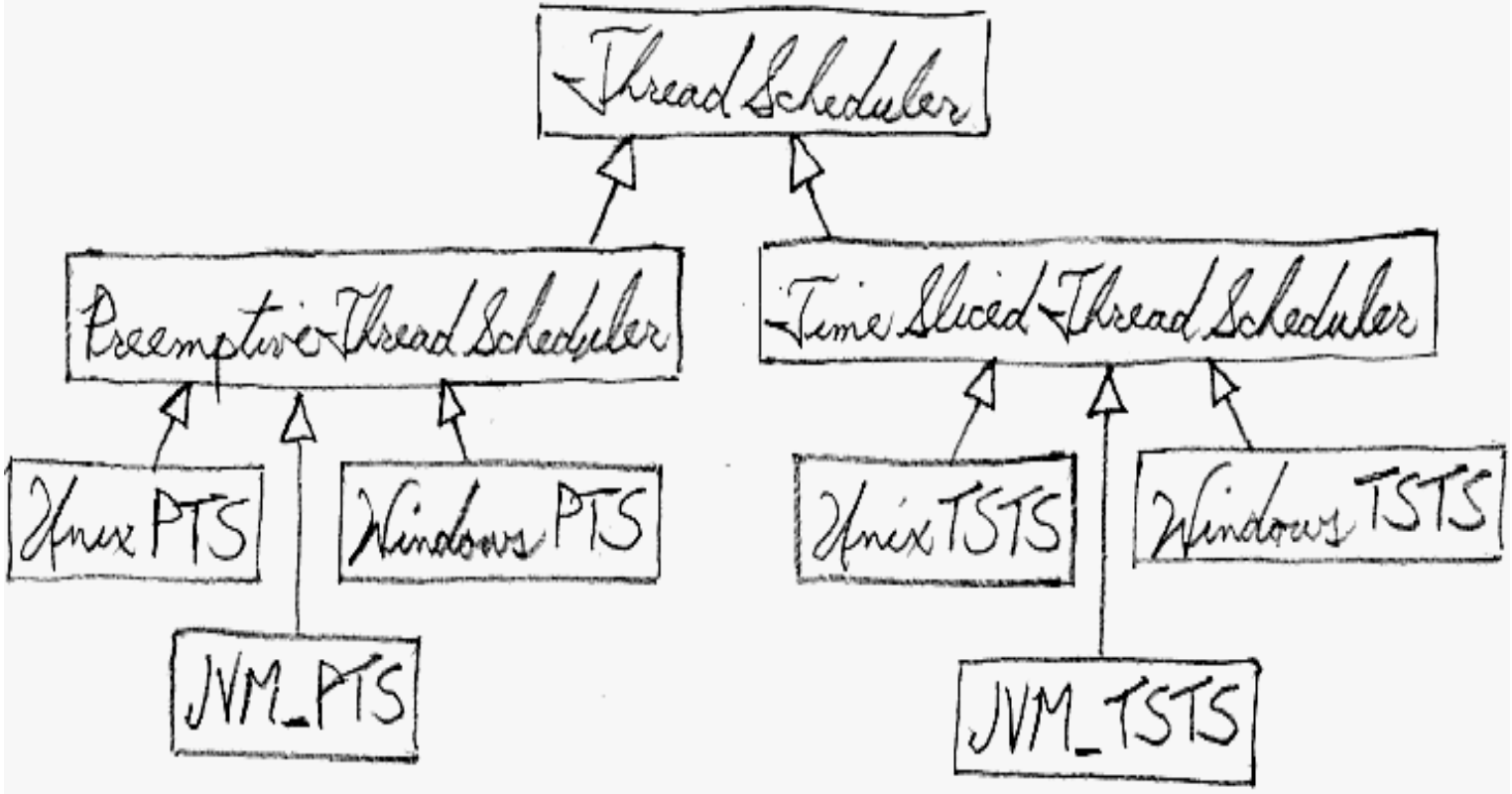
Consider the domain of "thread scheduling“:



Bridge

- There are two types of thread schedulers, and two types of operating systems or "platforms". Given this approach to specialization, we have to define a class for each permutation of these two dimensions. If we add a new platform (say ... Java's Virtual Machine), what would our hierarchy look like?

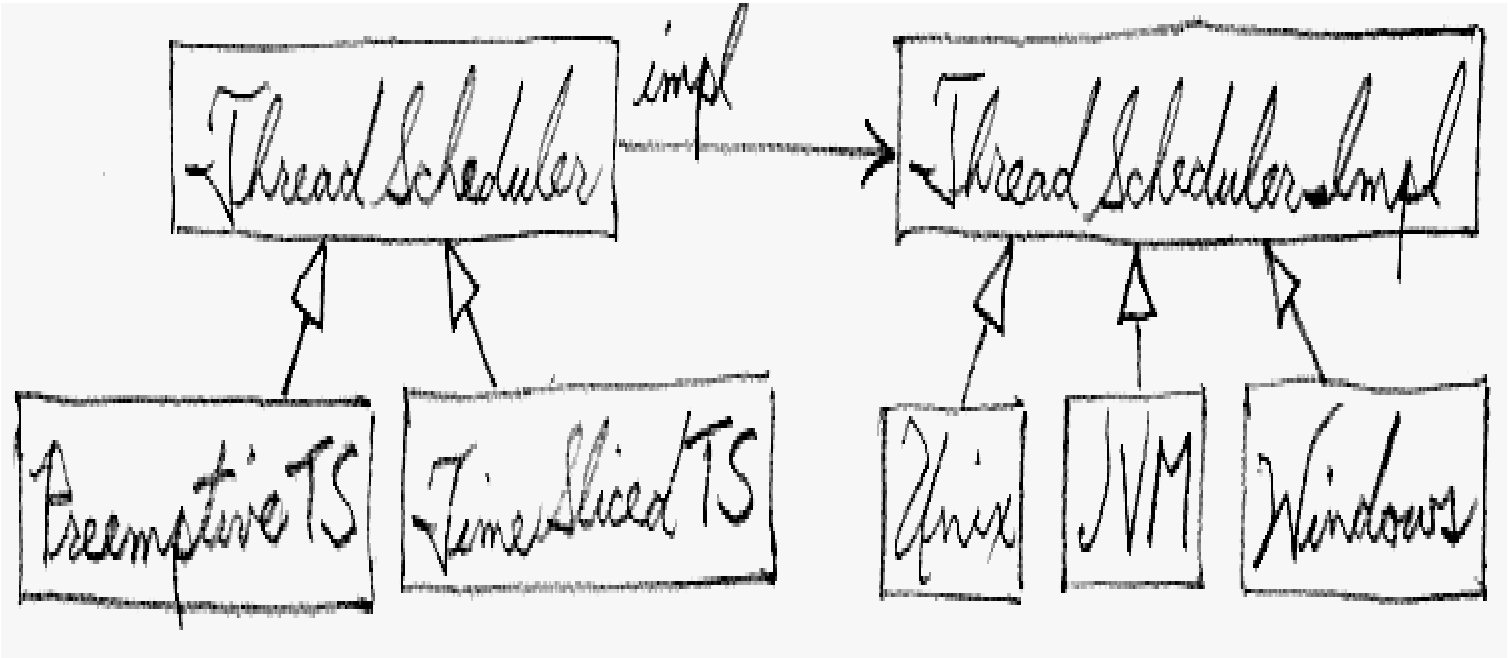
Bridge


Thread Scheduler
Preemptive Thread Scheduler
Time Sliced Thread Scheduler
Unix PTS
Windows PTS
JVM_PTS
Unix TSTS
Windows TSTS
JVM_TSTS

Bridge

- What if we had three kinds of thread schedulers, and four kinds of platforms? What if we had five kinds of thread schedulers, and ten kinds of platforms? The number of classes we would have to define is the product of the number of scheduling schemes and the number of platforms.
- The Bridge design pattern proposes refactoring this exponentially explosive inheritance hierarchy into two orthogonal hierarchies – one for platform-independent abstractions, and the other for platform-dependent implementations.

Bridge


ThreadScheduler
impl
ThreadSchedulerImpl
PreemptiveTS
TimeSlicedTS
Unix
JVM
Windows

Bridge

- Decompose the component's interface and implementation into orthogonal class hierarchies.
- The interface class contains a pointer to the abstract implementation class.
- This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class.
- The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.
- The interface object is the "handle" known and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared at run-time.

Factory Patterns

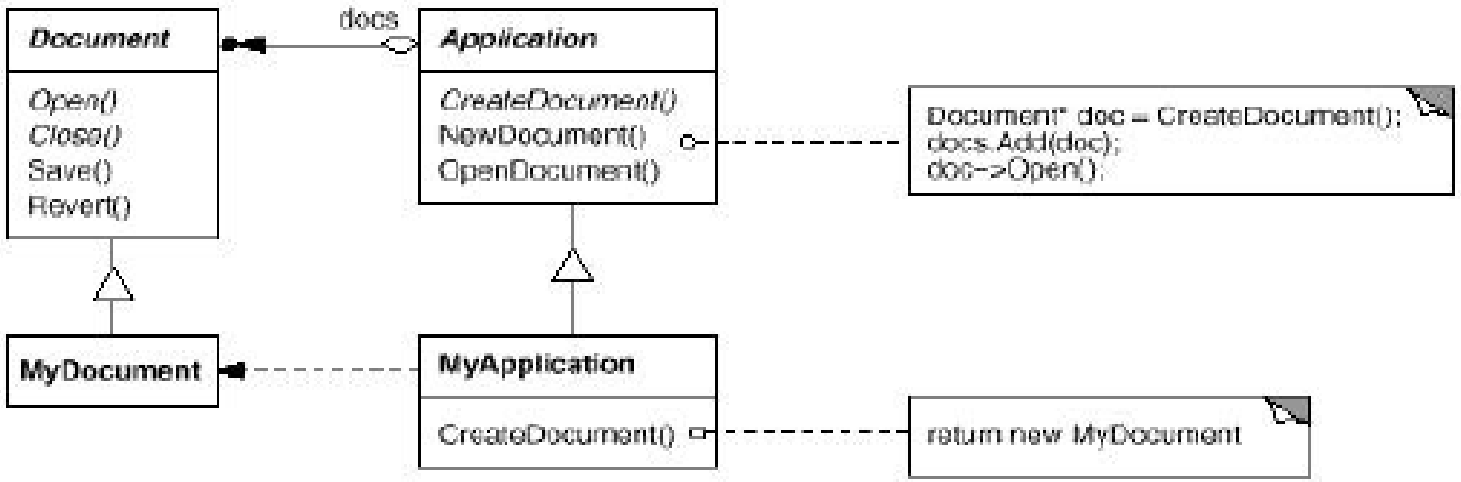
- Factory patterns are examples of creational patterns
- *Creational patterns* abstract the object instantiation process. They hide how objects are created and help make the overall system independent of how its objects are created and composed.
- *Class creational patterns* focus on the use of inheritance to decide the object to be instantiated
  - ⇨ Factory Method
- *Object creational patterns* focus on the delegation of the instantiation to another object
  - ⇨ Abstract Factory

Factory Patterns

- All OO languages have an idiom for object creation. In Java this idiom is the *new* operator. Creational patterns allow us to write methods that create new objects without explicitly using the new operator. This allows us to write methods that can instantiate different objects and that can be extended to instantiate other newly-developed objects, all without modifying the method's code! (Quick! Name the principle involved here!)

The Factory Method Pattern

- Intent
  - ⇒ Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Motivation
  - ⇒ Consider the following framework:



  - ⇒ The createDocument() method is a factory method.

The Factory Method Pattern

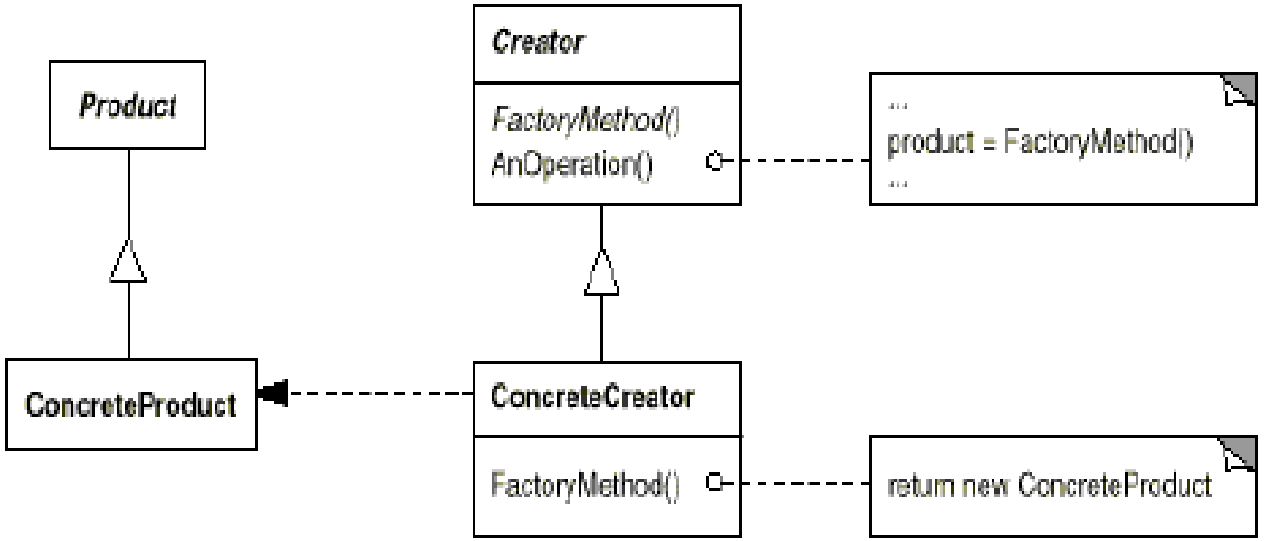
- Applicability

  Use the Factory Method pattern in any of the following situations:

  => A class can't anticipate the class of objects it must create

  => A class wants its subclasses to specify the objects it creates

- Structure



The Factory Method Pattern

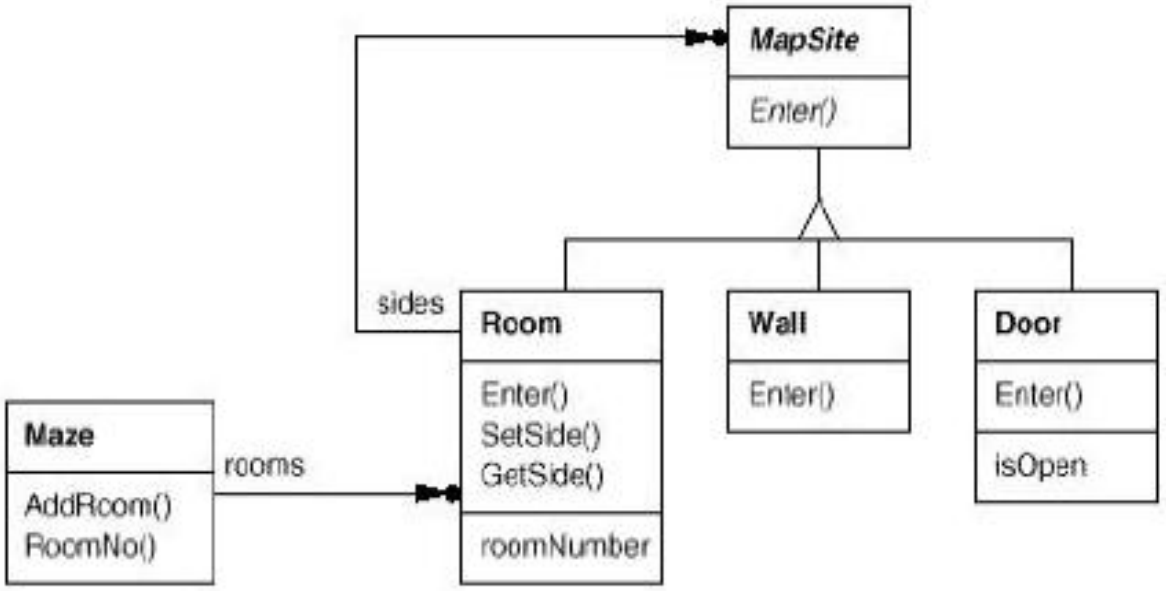
- Participants
  - Product
    - Defines the interface for the type of objects the factory method creates
  - ConcreteProduct
    - Implements the Product interface
  - Creator
    - Declares the factory method, which returns an object of type Product
  - ConcreteCreator
    - Overrides the factory method to return an instance of a ConcreteProduct
- Collaborations
  - Creator relies on its subclasses to implement the factory method so that it returns an instance of the appropriate ConcreteProduct

The Factory Method Pattern

- So what exactly does it mean when we say that "the Factory Method Pattern lets subclasses decide which class to instantiate?"
  - => It means that Creator class is written without knowing what actual ConcreteProduct class will be instantiated. The ConcreteProduct class which is instantiated is determined solely by which ConcreteCreator subclass is instantiated and used by the application.
  - => It does *not* mean that somehow the subclass decides at runtime which ConreteProduct class to create

Example 1

- Consider this maze game:



Example 1

- Here's a MazeGame class with a createMaze() method:

```
/**
 * MazeGame.
 */
public class MazeGame {

  // Create the maze.
  public Maze createMaze() {
    Maze maze = new Maze();
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door door = new Door(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
```

Example 1

```
        r1.setSide(MazeGame.North, new Wall());
        r1.setSide(MazeGame.East, door);
        r1.setSide(MazeGame.South, new Wall());
        r1.setSide(MazeGame.West, new Wall());
        r2.setSide(MazeGame.North, new Wall());
        r2.setSide(MazeGame.East, new Wall());
        r2.setSide(MazeGame.South, new Wall());
        r2.setSide(MazeGame.West, door);
        return maze;
    }

}
```

Example 1

- The problem with this createMaze() method is its *inflexibility*.
- What if we wanted to have enchanted mazes with EnchantedRooms and EnchantedDoors? Or a secret agent maze with DoorWithLock and WallWithHiddenDoor?
- What would we have to do with the createMaze() method? As it stands now, we would have to make significant changes to it because of the explicit instantiations using the *new* operator of the objects that make up the maze. How can we redesign things to make it easier for createMaze() to be able to create mazes with new types of objects?

Example 1

- Let's add factory methods to the MazeGame class:

```
/**
 * MazeGame with a factory methods.
 */
public class MazeGame {

   public Maze makeMaze() {return new Maze();}

   public Room makeRoom(int n) {return new Room(n);}

   public Wall makeWall() {return new Wall();}

   public Door makeDoor(Room r1, Room r2)
     {return new Door(r1, r2);}
```

Example 1

```
  public Maze createMaze() {
    Maze maze = makeMaze();
    Room r1 = makeRoom(1);
    Room r2 = makeRoom(2);
    Door door = makeDoor(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
    r1.setSide(MazeGame.North, makeWall());
    r1.setSide(MazeGame.East, door);
    r1.setSide(MazeGame.South, makeWall());
    r1.setSide(MazeGame.West, makeWall());
    r2.setSide(MazeGame.North, makeWall());
    r2.setSide(MazeGame.East, makeWall());
    r2.setSide(MazeGame.South, makeWall());
    r2.setSide(MazeGame.West, door);
    return maze;
  }
}
```

Example 1

- We made createMaze() just slightly more complex, but a lot more flexible!
- Consider this EnchantedMazeGame class:

```
public class EnchantedMazeGame extends MazeGame {
  public Room makeRoom(int n) {return new EnchantedRoom(n);}
  public Wall makeWall() {return new EnchantedWall();}
  public Door makeDoor(Room r1, Room r2)
    {return new EnchantedDoor(r1, r2);}
}
```

- The createMaze() method of MazeGame is inherited by EnchantedMazeGame and can be used to create regular mazes or enchanted mazes *without modification*!

Example 1

- The reason this works is that the createMaze() method of MazeGame defers the creation of maze objects to its subclasses. That's the Factory Method pattern at work!
- In this example, the correlations are:
  - Creator => MazeGame
  - ConcreteCreator => EnchantedMazeGame (MazeGame is also a ConcreteCreator)
  - Product => MapSite
  - ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor

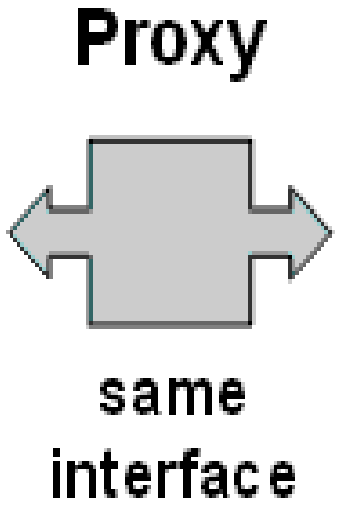
The Factory Method Pattern

- Consequences
  - ⇨ Benefits
    - Code is made more flexible and reusable by the elimination of instantiation of application-specific classes
    - Code deals only with the interface of the Product class and can work with any ConcreteProduct class that supports this interface
  - ⇨ Liabilities
    - Clients might have to subclass the Creator class just to instantiate a particular ConcreteProduct
- Implementation Issues
  - ⇨ Creator can be abstract or concrete
  - ⇨ Should the factory method be able to create multiple kinds of products? If so, then the factory method has a parameter (possibly used in an if-else!) to decide what object to create.

Proxy

- **Intent**
- Provide a surrogate or placeholder for another object to control access to it. [GoF, p207]
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.
- **Problem**
- You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

Proxy

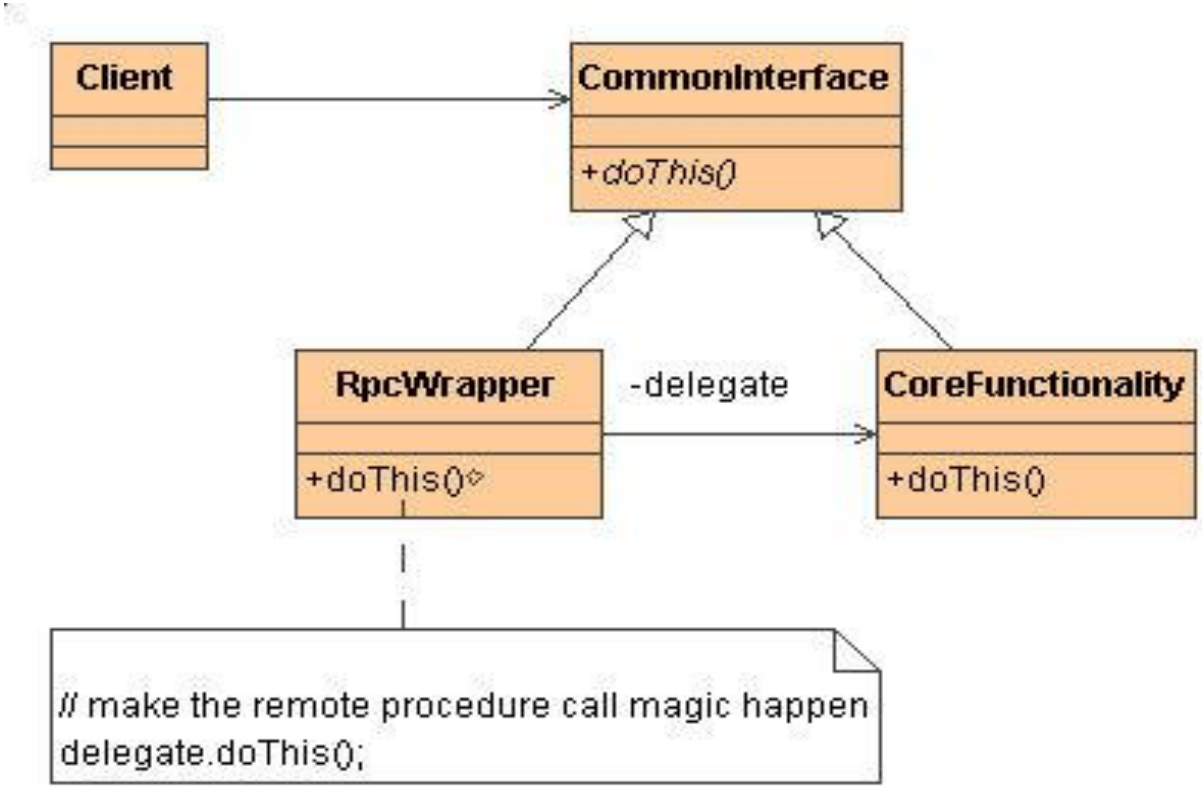


1) Take a class with desirable functionality and wrap it in a new class that provides the same interface and additional functionality (e.g. distributed, controlled, or monitored access).

Proxy

- There are four common situations in which the Proxy pattern is applicable.
- A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
- A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.
- A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.
- A smart proxy interposes additional actions when an object is accessed. Typical uses include:
  - Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),
  - Loading a persistent object into memory when it's first referenced,
  - Checking that the real object is locked before it is accessed to ensure that no other object can change it.

Proxy

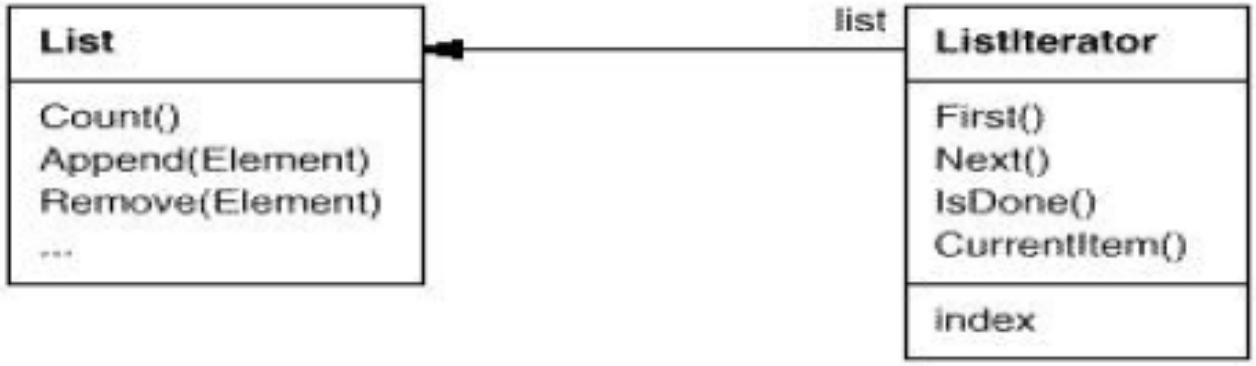

Client
CommonInterface
+doThis()
RpcWrapper
-delegate
CoreFunctionality
+doThis()
+doThis()
// make the remote procedure call magic happen
delegate.doThis();

The Iterator Pattern

- Intent
  - ⇒ Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
  - ⇒ An *aggregate object* is an object that contains other objects for the purpose of grouping those objects as a unit. It is also called a *container* or a *collection*. Examples are a linked list and a hash table.
- Also Known As
  - ⇒ Cursor
- Motivation
  - ⇒ An aggregate object such as a list should allow a way to traverse its elements without exposing its internal structure
  - ⇒ It should allow different traversal methods
  - ⇒ It should allow multiple traversals to be in progress concurrently
  - ⇒ But, we really do not want to add all these methods to the interface for the aggregate

The Iterator Pattern

- List aggregate with iterator:



The Iterator Pattern

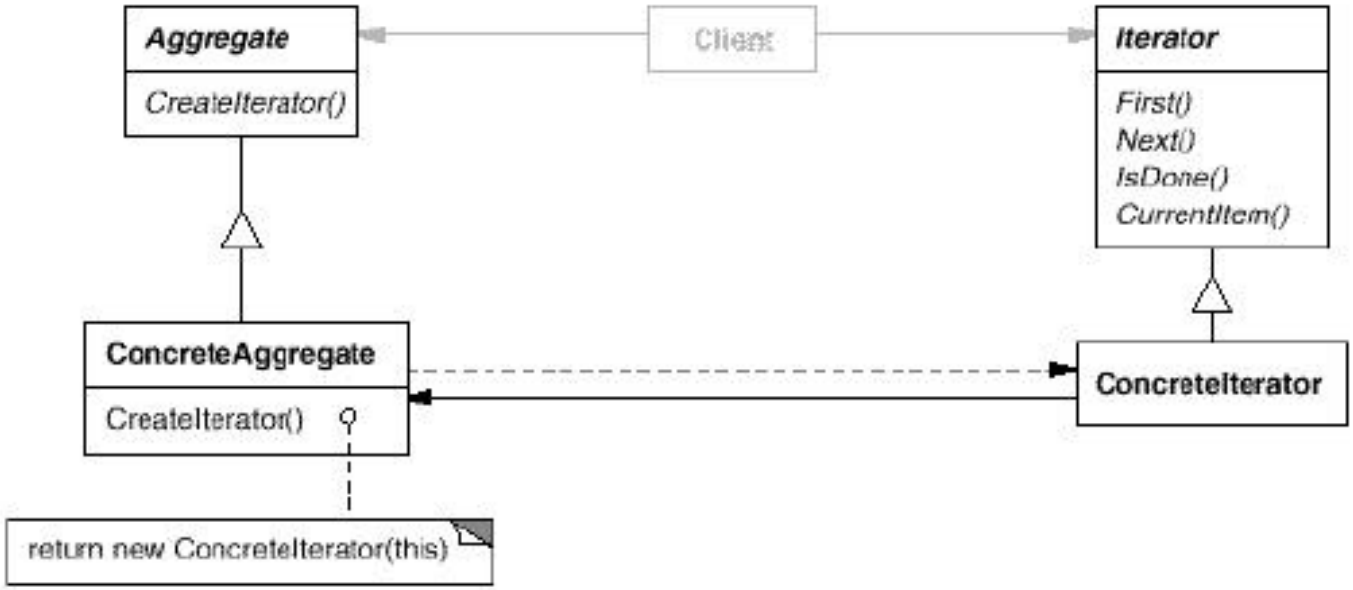
- Applicability

  Use the Iterator pattern:

  - ⇨ To support traversals of aggregate objects without exposing their internal representation
  - ⇨ To support multiple, concurrent traversals of aggregate objects
  - ⇨ To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration)

The Iterator Pattern

- Structure



The Iterator Pattern

- Participants
  - Iterator
    - Defines an interface for accessing and traversing elements
  - ConcreteIterator
    - Implements the Iterator interface
    - Keeps track of the current position in the traversal
  - Aggregate
    - Defines an interface for creating an Iterator object (a factory method!)
  - ConcreteAggregate
    - Implements the Iterator creation interface to return an instance of the proper ConcreteIterator

The Iterator Pattern

- Consequences
  - ⇨ Benefits
    - Simplifies the interface of the Aggregate by not polluting it with traversal methods
    - Supports multiple, concurrent traversals
    - Supports variant traversal techniques
  - ⇨ Liabilities
    - None!

The Iterator Pattern

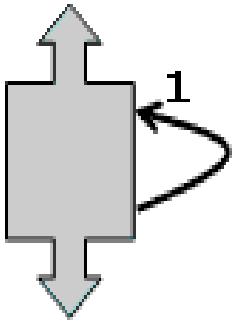
- Known Uses
  - ObjectSpace's Java Generic Library containers
  - Rogue Wave's Tools.h++ collections
  - java.util.Enumeration interface
  - Java 2 Collections Framework Iterator interface
- Related Patterns
  - Factory Method
    - Polymorphic iterators use factory methods to instantiate the appropriate iterator subclass
  - Composite
    - Iterators are often used to recursively traverse composite structures

Decorator

- **Intent**
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. [GoF, p175]
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.
- **Problem**
- You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

Decorator

Decorator



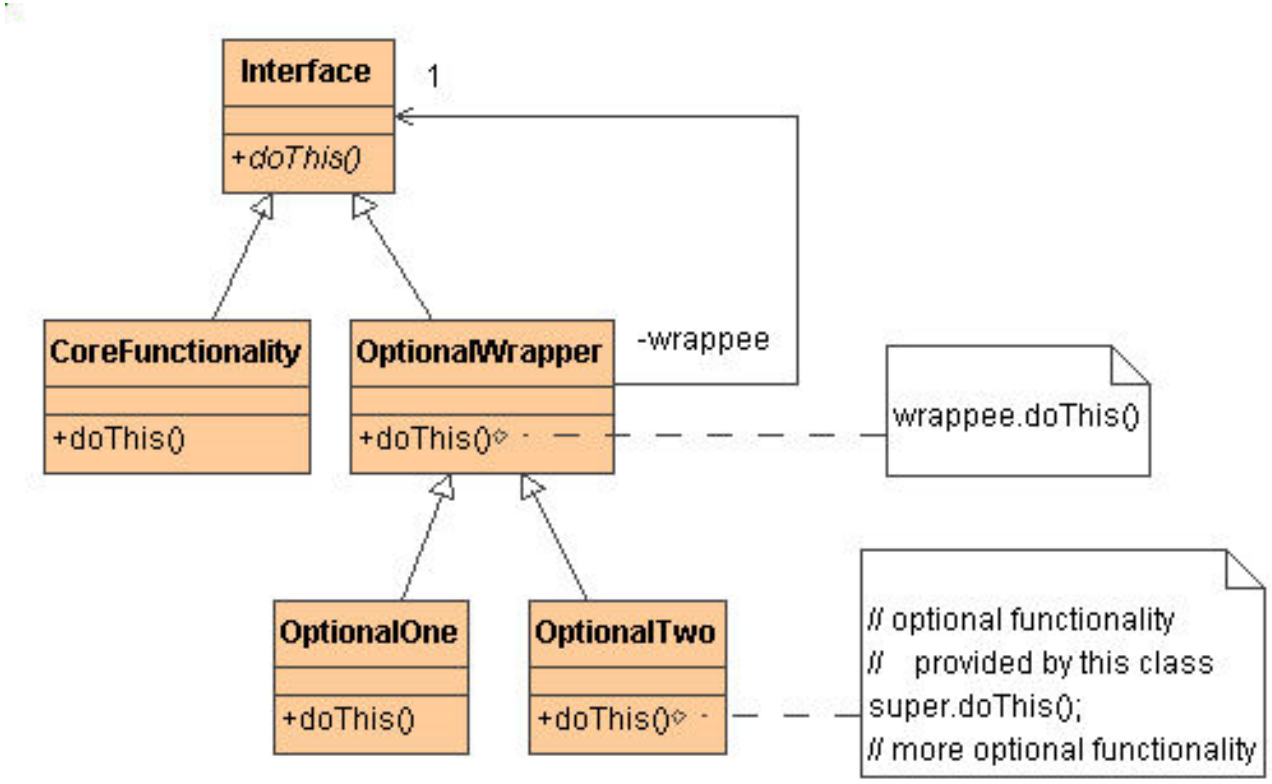
1) Identify if the client would like to dynamically and arbitrarily embellish a "core" object.

2) Start with the Composite design, and change the 1-to-many recursive relationship to 1-to-1.

3) Define each "embellishment" class as a derived class of the class whose role has morphed from "Composite" to "Decorator".

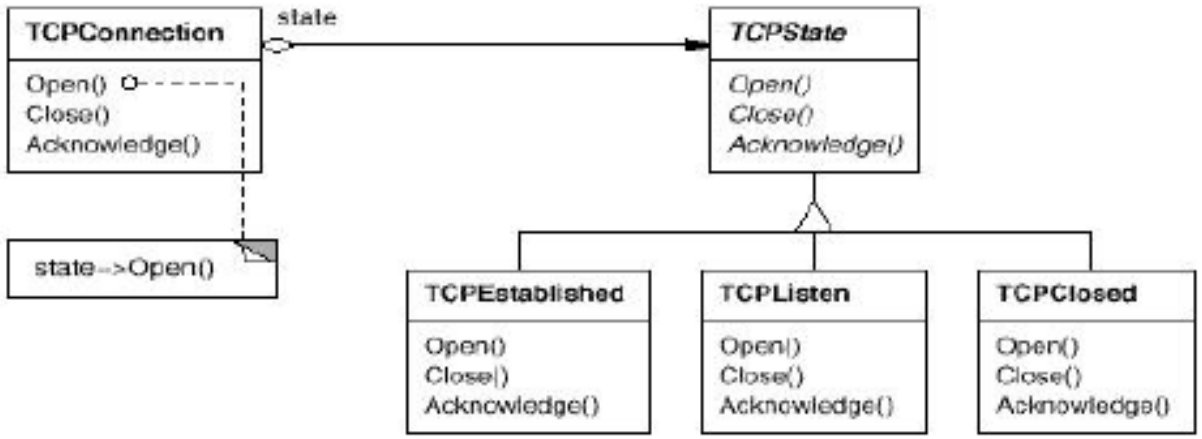
4) The client can now wrap a "core" object with any number of Decorator objects. Each Decorator contributes its "embellishment" functionality, and then delegates to its wrappee.

Decorator


Interface
1
+doThis()
CoreFunctionality
+doThis()
OptionalWrapper
-wrappee
+doThis()
wrappee.doThis()
OptionalOne
+doThis()
OptionalTwo
+doThis()
// optional functionality
// provided by this class
super.doThis();
// more optional functionality

The State Pattern

- Intent
  - ⇒ Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Motivation



The State Pattern

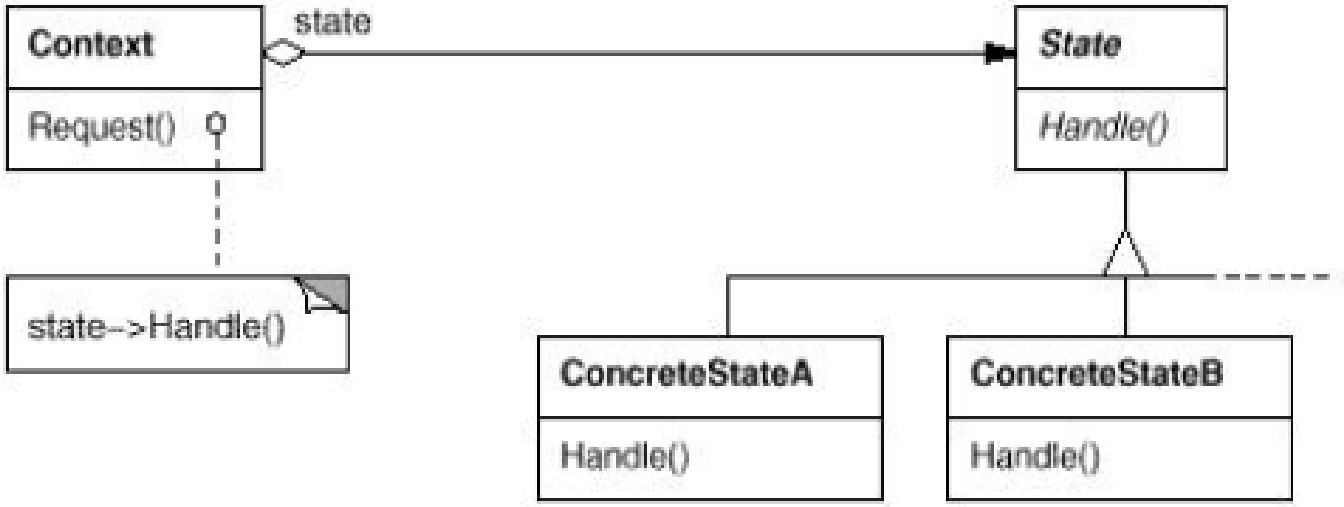
- Applicability

  Use the State pattern whenever:

  - ⇒ An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
  - ⇒ Operations have large, multipart conditional statements that depend on the object's state. The State pattern puts each branch of the conditional in a separate class.

The State Pattern

- Structure

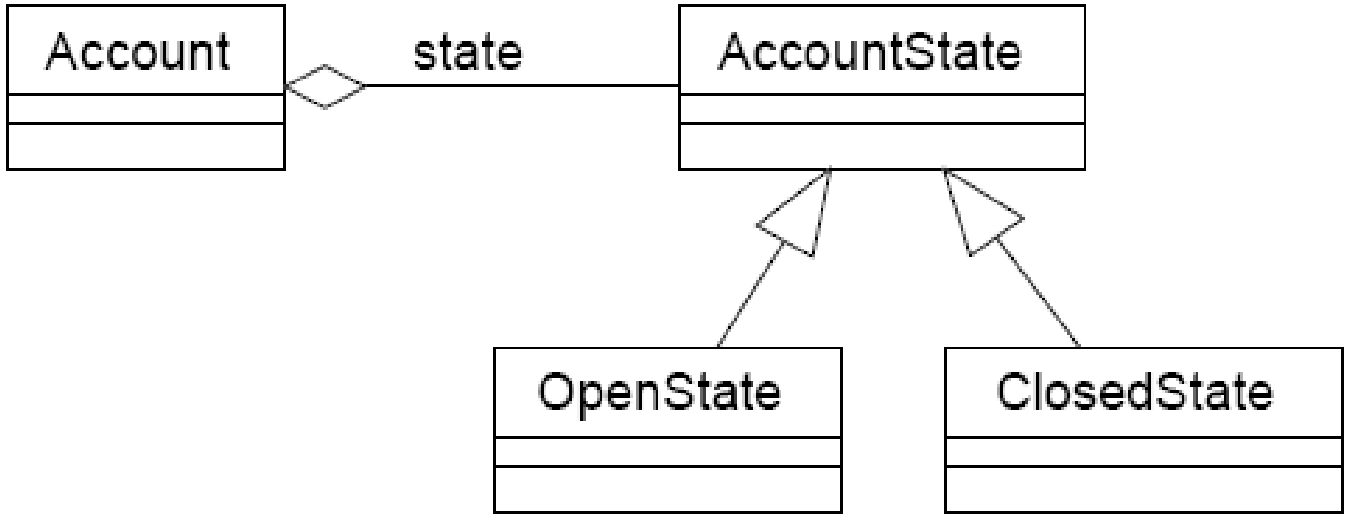


The State Pattern

- Consequences
  - ⇨ Benefits
    - Puts all behavior associated with a state into one object
    - Allows state transition logic to be be incorporated into a state object rather than in a monolithic if or switch statement
    - Helps avoid inconsistent states since state changes occur using just the one state object and not several objects or attributes
  - ⇨ Liabilities
    - Increased number of objects

Example 1

- Situation: A bank account can change from an open account to a closed account and back to an open account again. The behavior of the two types of accounts is different.
- Solution: Use the State pattern!



Example 2 - SPOP

- Consider a simplified version of the Post Office Protocol used to download e-mail from a mail server
- Simple POP (SPOP) supports the following command:
  - ⇒ USER username
    - The USER command with a username must be the first command issued
  - ⇒ PASS password
    - The PASS command with a password or the QUIT command must come after USER. If the username and password are valid, then the user can use other commands.
  - ⇒ LIST <message number>
    - The LIST command returns the size of all messages in the mail box. If the optional message number is specified, then it returns the size of that message.

Example 2 – SPOP (Continued)

- RETR <message number>
  - The RETR command retrieves all message in the mail box. If the optional message number is specified, then it retrieves that message.
- QUIT
  - The QUIT command updates the mail box to reflect transactions taken, then logs the user out.

Example 2 – SPOP (Continued)

- Here's a version of an SPop class without using the State pattern:

```
public class SPop {
  static final int QUIT = 1;
  static final int HAVE_USER_NAME = 2;
  static final int START = 3;
  static final int AUTHORIZED = 4;
  private int state = START;
  String userName;
  String password;
```

Example 2 – SPOP (Continued)

```
public void user(String userName) {
  switch (state) {
    case START: {
      this.userName = userName;
      state = HAVE_USER_NAME;
      break;
    }
    default: { // Invalid command
      sendErrorMessageOrWhatEver();
      endLastSessionWithoutUpdate();
      userName = null;
      password = null;
      state = START;
    }
  }
}
```

Example 2 – SPOP (Continued)

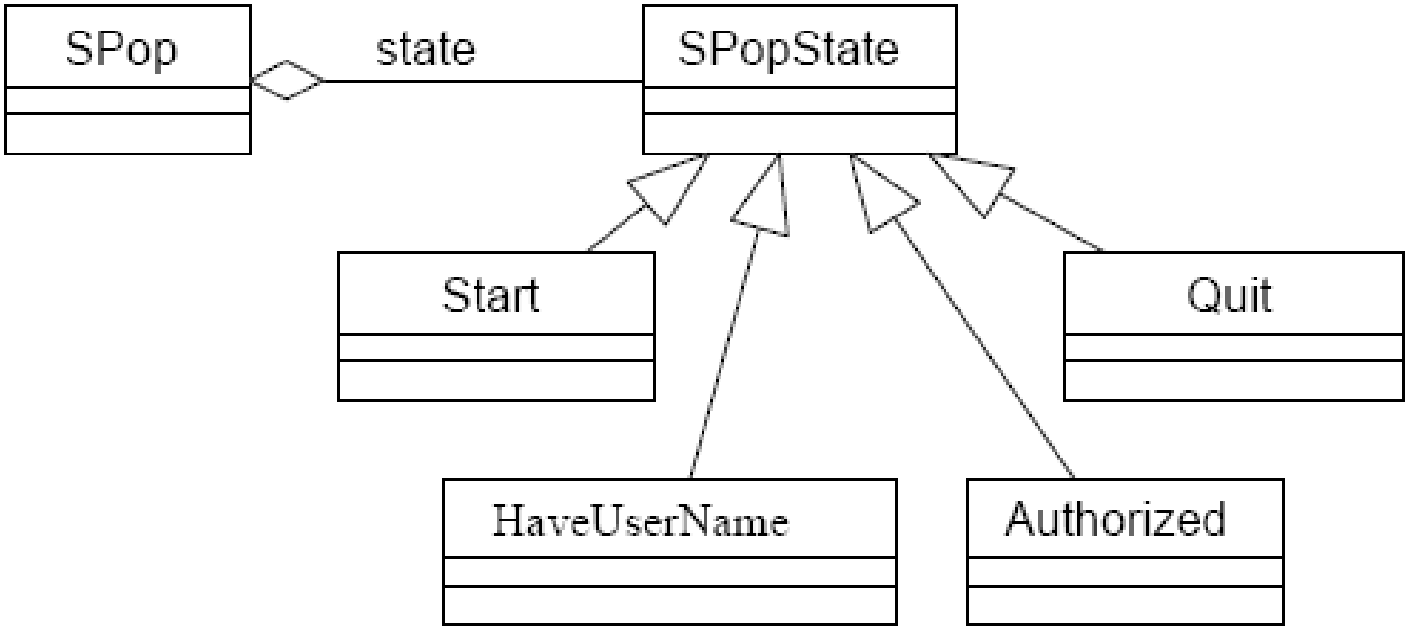
```
public void pass(String password) {
  switch (state) {
    case HAVE_USER_NAME: {
      this.password = password;
      if (validateUser())
        state = AUTHORIZED;
      else {
        sendErrorMessageOrWhatEver();
        userName = null;
        password = null;
        state = START;
      }
    }
```

Example 2 – SPOP (Continued)

```
        default: {  // Invalid command
          sendErrorMessageOrWhatEver();
          endLastSessionWithoutUpdate();
          state = START;
        }
      }
    }
    ...
}
```

Example 2 – SPOP (Continued)

- Now let's use the State pattern!
- Here's the class diagram:



Example 2 – SPOP (Continued)

- First, we'll define the SPopState class. Notice that this class is a concrete class that defines default actions.

```
public class SPopState {

  public SPopState user(String userName) {default action here}

  public SPopState pass(String password) {default action here}

  public SPopState list(int messageNumber) {default action here}

  public SPopState retr(int messageNumber) {default action here}

  public SPopState quit() {default action here}

}
```

Example 2 – SPOP (Continued)

- Here's the Start class:

```
public class Start extends SPopState {

  public SPopState user(String userName) {
    return new HaveUserName(userName);
  }

}
```

Example 2 – SPOP (Continued)

- Here's the HaveUserName class:

```
public class HaveUserName extends SPopState {

  String userName;

  public HaveUserName(String userName) {
    this.userName = userName;
  }

  public SPopState pass(String password) {
    if (validateUser(userName, password)
      return new Authorized(userName);
    else
      return new Start();
  }
}
```

Example 2 – SPOP (Continued)

- Finally, here is the SPop class that uses these state classes:

```
public class SPop {
  private SPopState state = new Start();

  public void user(String userName) {
    state = state.user(userName);
  }

  public void pass(String password) {
    state = state.pass(password);
  }

  public void list(int messageNumber) {
    state = state.list(messageNumber);
  }
  ...
}
```

Example 2 – SPOP (Continued)

- Note, that in this example, the state classes specify the next state
- We could have the SPop class itself determine the state transition (the state classes now return true of false):

```
public class SPop {
  private SPopState state = new Start();
  public void user(String userName) {
    state.user(userName);
    state = new HaveUserName(userName);
  }
  public void pass(String password) {
    if (state.pass(password))
      state = new Authorized();
    else
      state = new Start();
  }
}
```

Example 2 – SPOP (Continued)

- Multiple instances of SPop could share state objects if the state objects have no required instance variables or the state objects store their instance variables elsewhere
- Such sharing of objects is an example of the Flyweight Pattern
- How can the state object store its state elsewhere?
  - => Have the Context store this data and pass it to the state object (a push model)
  - => Have the Context store this data and have the state object retrieve it when needed (a pull model)

Example 2 – SPOP (Continued)

- Here's an example of the Context storing the state and passing it to the state objects:

```
public class SPop {
  private SPopState state = new Start();
  String userName;
  String password;

  public void user(String newName) {
    this.userName = newName;
    state.user(newName);
  }

  public void pass(String password) {
    state.pass(userName, password);
  }
  ...
```

Example 2 – SPOP (Continued)

- Here the Context stores the state and the state objects retrieve it:

```
public class SPop {
  private SPopState state = new Start();
  String userName;
  String password;

  public String getUserName() {return userName;}

  public String getPassword() {return password;}

  public void user(String newName) {
    this.userName = newName ;
    state.user(this);
  }
  ...
}
```

Example 2 – SPOP (Continued)

- And here is how the HaveUserName state object retrieves the state in its user() method:

```
public class HaveUserName extends SPopState {

  public SPopState user(SPop mailServer) {
    String userName = mailServer.getUserName();
    ...
  }
  ...
}
```

State Vs. Strategy

- Note the similarities between the State and Strategy patterns! The difference is one of intent.
  - ⇨ A State object encapsulates a state-dependent behavior (and possibly state transitions)
  - ⇨ A Strategy object encapsulates an algorithm
- And they are both examples of Composition with Delegation!

The Builder pattern

Type & intent

- One of the Creational Pattern
- Intent:
  - Separates the construction of a complex object from its representation so that the same construction process can create different representations.

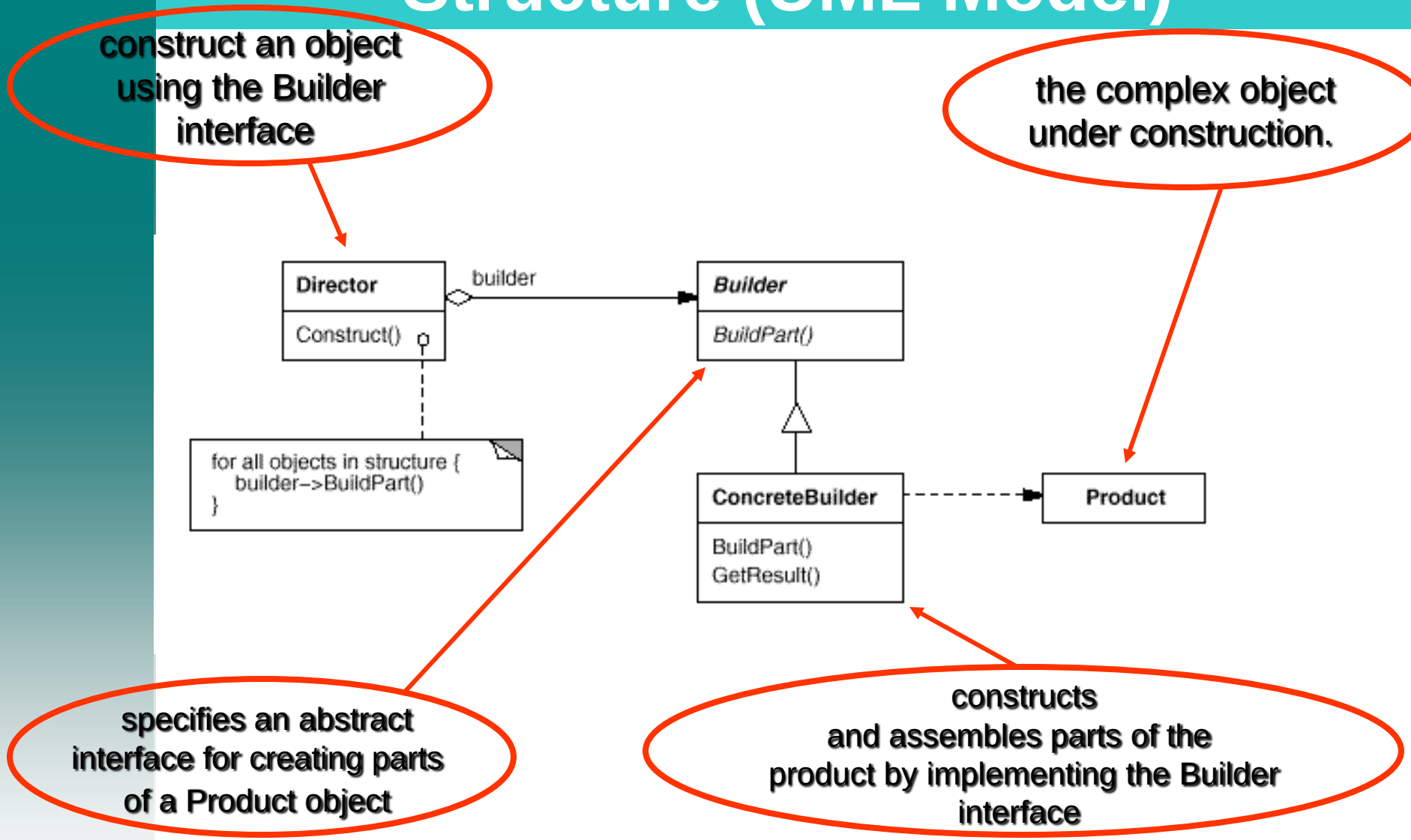
Application

- The builder pattern is an object creation software design pattern.
- Unlike the factory method / abstract factory pattern and the factory method pattern whose intention is to enable polymorphism, the intention of the builder pattern is to find a solution to the telescoping constructor anti-pattern.
- The telescoping constructor anti-pattern occurs when the increase of object constructor parameter combination leads to an exponential list of constructors.
- Instead of using numerous constructors, the builder pattern uses another object, a builder, that receives each initialization parameter step by step and then returns the resulting constructed object at once.

Applicability

- The Builder pattern assembles a number of objects in various ways depending on the data.
- Use the Builder pattern when
  - the algorithm for creating a complex object should be independent on the parts that make up the object and how they're assembled.
  - the construction process must allow different representations for the object that's constructed.

Structure (UML Model)



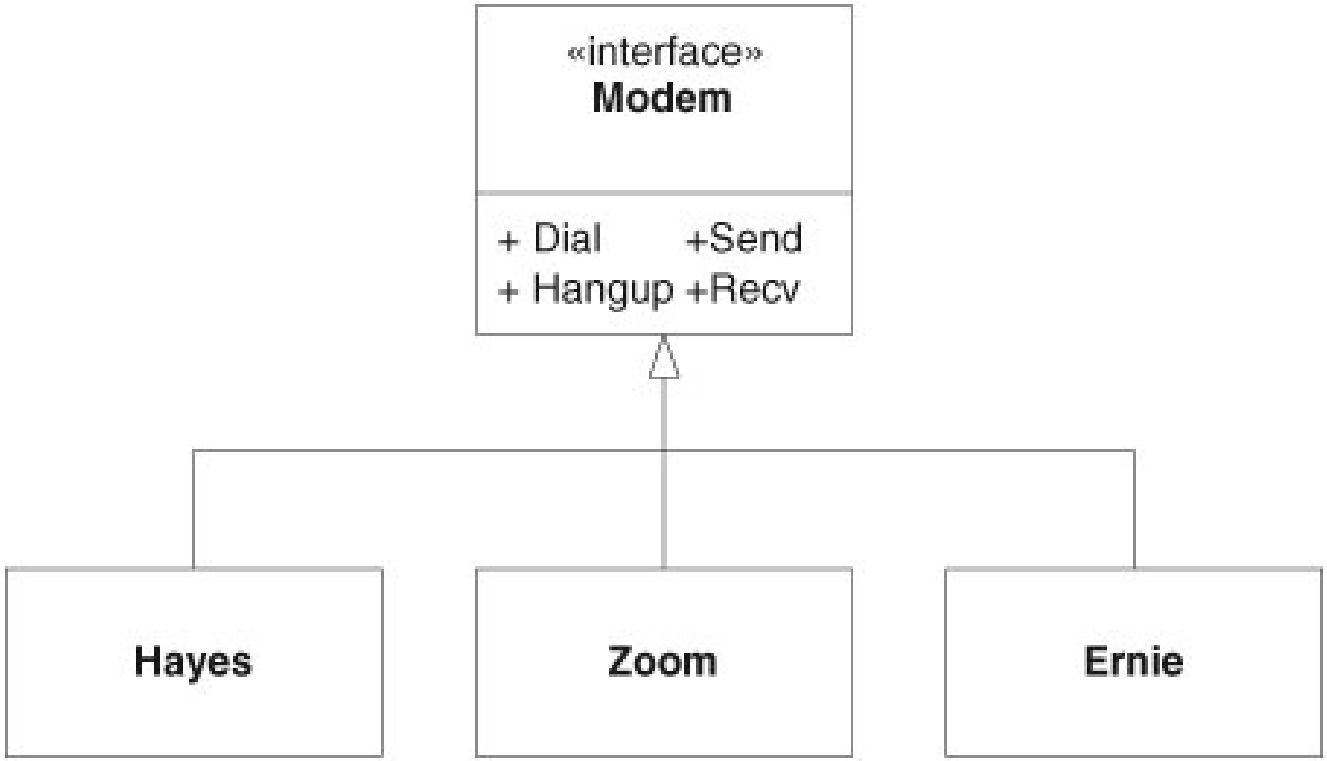
Participants

- **Builder:** specifies an abstract interface for creating parts of a Product object.
- **ConcreteBuilder:**
  - constructs and assembles parts of the product by implementing the Builder interface.
  - Defines and keeps track of the representation it creates.
  - Provides an interface for retrieving the product.
- **Director:** constructs an object using the Builder interface.
- **Product:** represents the complex object under construction.

Consequences

- Abstracts the construction implementation details of a class type. It lets you vary the internal representation of the product that it builds.
- Encapsulates the way in which objects are constructed improving the modularity of a system.
- Finer control over the creation process, by letting a builder class have multiple methods that are called in a sequence to create an object.
- Each specific Builder is independent of any others.

Visitor



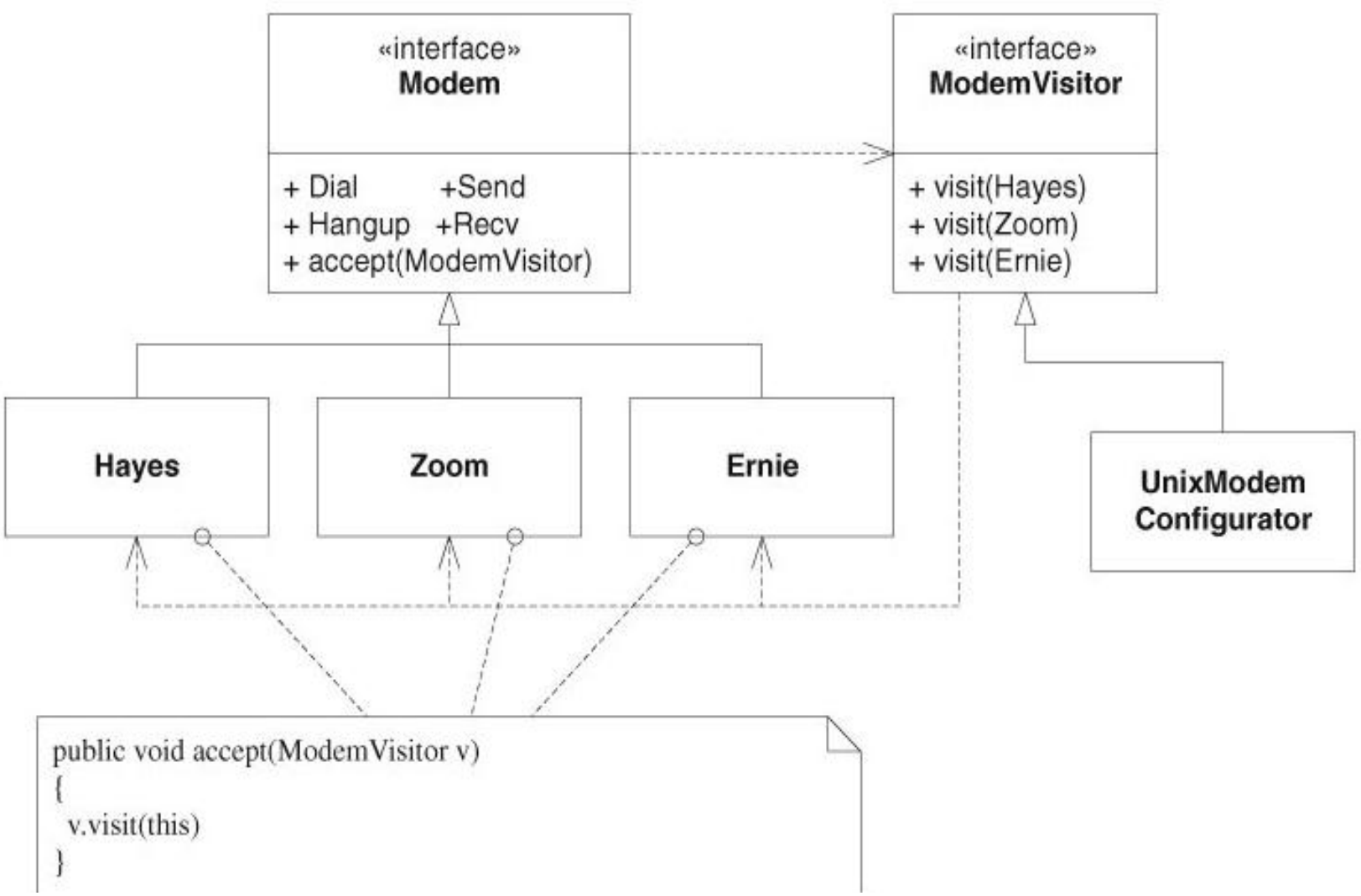
Visitor

- You need to add a new method to a hierarchy of classes, but the act of adding it will be painful or damaging to the design.
  - This is a common problem.
- The base class has the generic methods common to all modems.
- The derivatives represent the drivers for many different modem manufacturers and types.
- Suppose also that you have a requirement to add a new method, named configureForUnix, to the hierarchy. This method will configure the modem to work with the UNIX operating system. The method will do something different in each modem derivative, because each modem has its own particular idiosyncrasies for setting its configuration, and dealing with UNIX.

Visitor

- What about Windows, what about OSX, what about Linux?
  - Must we really add a new method to the Modem hierarchy for every new operating system that we use?
  - Clearly, this is ugly. We'll never be able to close the Modem interface. Every time a new operating system comes along, we'll have to change that interface and redeploy all the modem software.

Visitor


«interface»
Modem
+ Dial +Send
+ Hangup +Recv
+ accept(ModemVisitor)
«interface»
ModemVisitor
+ visit(Hayes)
+ visit(Zoom)
+ visit(Ernie)
Hayes
Zoom
Ernie
UnixModem
Configurator
public void accept(ModemVisitor v)
{
v.visit(this)
}

Visitor

- Note that the visitor hierarchy has a method in for every derivative of the visited (Modem) hierarchy.
  - This is a kind of 90° rotation: from derivatives to methods.
- The test code shows that to configure a modem for UNIX, a programmer creates an instance of the UnixModemConfigurator class and passes it to the Accept function of the Modem.
- The appropriate Modem derivative will then call Visit(this) on Modem-Visitor, the base class of UnixModemConfigurator.
  - If that derivative is a Hayes, Visit(this) will call public void Visit(Hayes), which will deploy to the public void Visit(Hayes) function in UnixModemConfigurator, which then configures the Hayes modem for Unix.

Visitor

```
public interface Modem
{
  void Dial(string pno);
  void Hangup();
  void Send(char c);
  char Recv();
  void Accept(ModemVisitor v);
}
```

Visitor

```
public class HayesModem implements Modem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v)
   {v.Visit(this);}

  public string configurationString = null;
}
```

Visitor

```
public class UnixModemConfigurator implements ModemVisitor
{
  public void Visit(HayesModem m)
  {
    m.configurationString = "&s1=4&D=3";
  }

  public void Visit(ZoomModem m)
  {
    m.configurationValue = 42;
  }
  public void Visit(ErnieModem m)
  {
    m.internalPattern = "C is too slow";
  }
}
```

Visitor

- Having built this structure, new operating system configuration functions can be added by adding new derivatives of ModemVisitor without altering the Modem hierarchy in any way.
  - So the VISITOR pattern substitutes derivatives of ModemVisitor for methods in the Modem hierarchy.
- This dual dispatch involves two polymorphic dispatches.
  - The first is the Accept function, which resolves the type of the object that Accept is called on. The second dispatchthe Visit method called from the resolved Accept methodresolves to the particular function to be executed.
- The two dispatches of VISITOR form a matrix of functions.
  - In our modem example, one axis of the matrix is the various types of modems; the other axis, the various types of operating systems. Every cell in this matrix is filled in with a function that describes how to initialize the particular modem for the particular operating system.

Acyclic Visitor

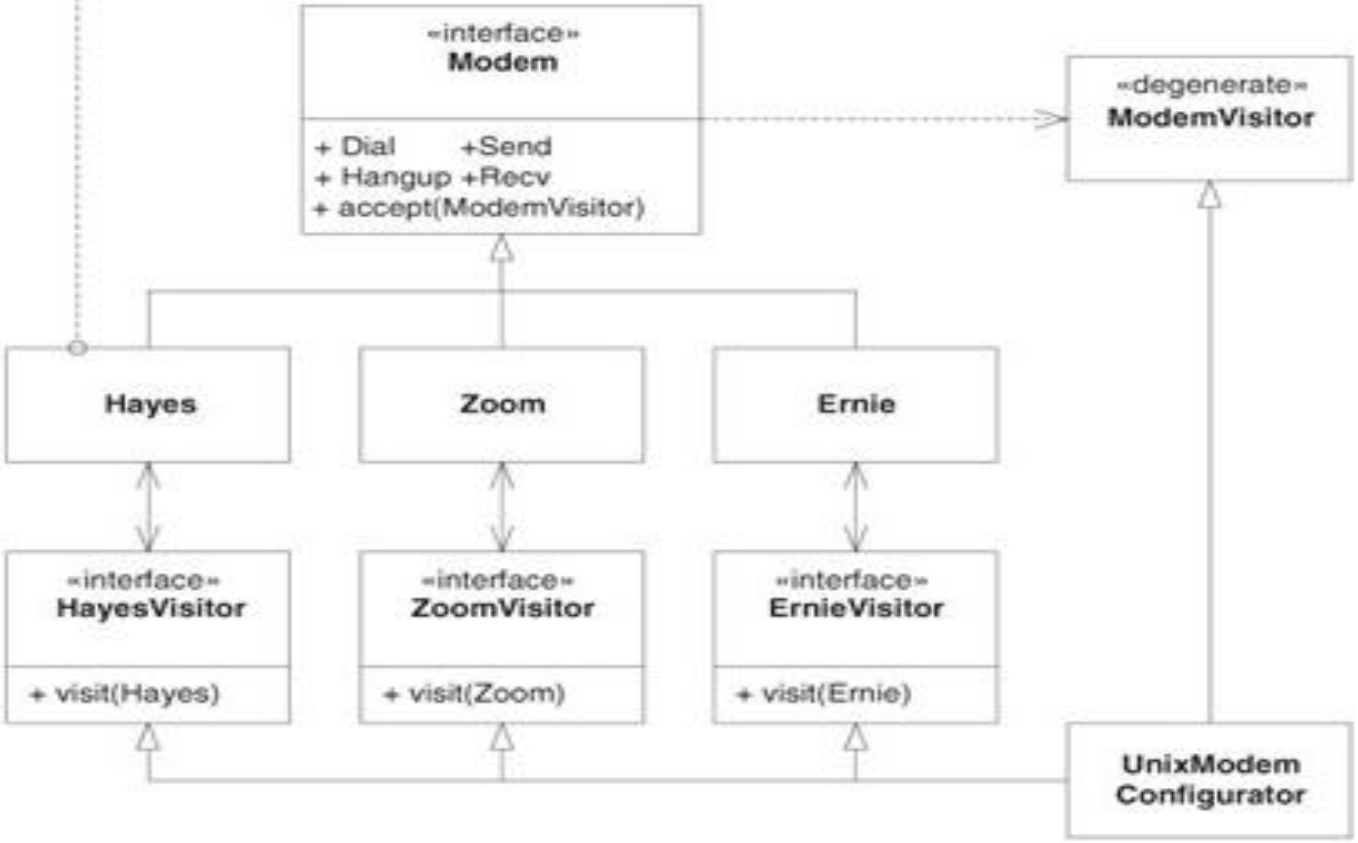
- Note that the base class of the visited (Modem) hierarchy depends on the base class of the visitor hierarchy (ModemVisitor).
- Note also that the base class of the visitor hierarchy has a function for each derivative of the visited hierarchy. This cycle of dependencies ties all the visited derivativesall the modemstogether, making difficult to compile the visitor structure incrementally or to add new derivatives to the visited hierarchy.
- The VISITOR pattern works well in programs in which the hierarchy to be modified does not need new derivatives very often.
- If Hayes, Zoom, and Ernie were the only Modem derivatives that were likely to be needed or if the incidence of new Modem derivatives was expected to be infrequent, VISITOR would be appropriate.

Acyclic Visitor

- On the other hand, if the visited hierarchy is highly volatile, such that many new derivatives will need to be created, the visitor base class (e.g., ModemVisitor) will have to be modified and recompiled along with all its derivatives every time a new derivative is added to the visited hierarchy.
- ACYCLIC VISITOR can be used to solve these problems.
- This variation breaks the dependency cycle by making the Visitor base class (ModemVisitor) degenerate, that is, without methods. Therefore, this class does not depend on the derivatives of the visited hierarchy.

Acyclic Visitor

```
public void accept(ModemVisitor v) {
  try {
    HayesVisitor hv = (HayesVisitor) v;
    hv.visit(this);
  }
  catch (InvalidCastException e) {}
}
```



Acyclic Visitor

- The visitor derivatives also derive from visitor interfaces.
- There is one visitor interface for each derivative of the visited hierarchy.
- This is a 180° rotation from derivatives to interfaces.
- The Accept functions in the visited derivatives cast the visitor base class to the appropriate visitor interface.
  - If the cast succeeds, the method invokes the appropriate visit function.

Acyclic Visitor

```
public interface Modem
{
  void Dial(string pno);
  void Hangup();
  void Send(char c);
  char Recv();
  void
    Accept(ModemVisitor
    v);
}
```

```
public interface
    ModemVisitor
{
}
public interface
    ErnieModemVisitor
    extends ModemVisitor
{
  void Visit(ErnieModem
    m);
}
```

Acyclic Visitor

```
public class ErnieModem implements Modem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v)
  {
    if(v instanceofErnieModemVisitor)
      ((ErnieModemVisitor) v).Visit(this);
  }

  public string internalPattern = null;
}
```

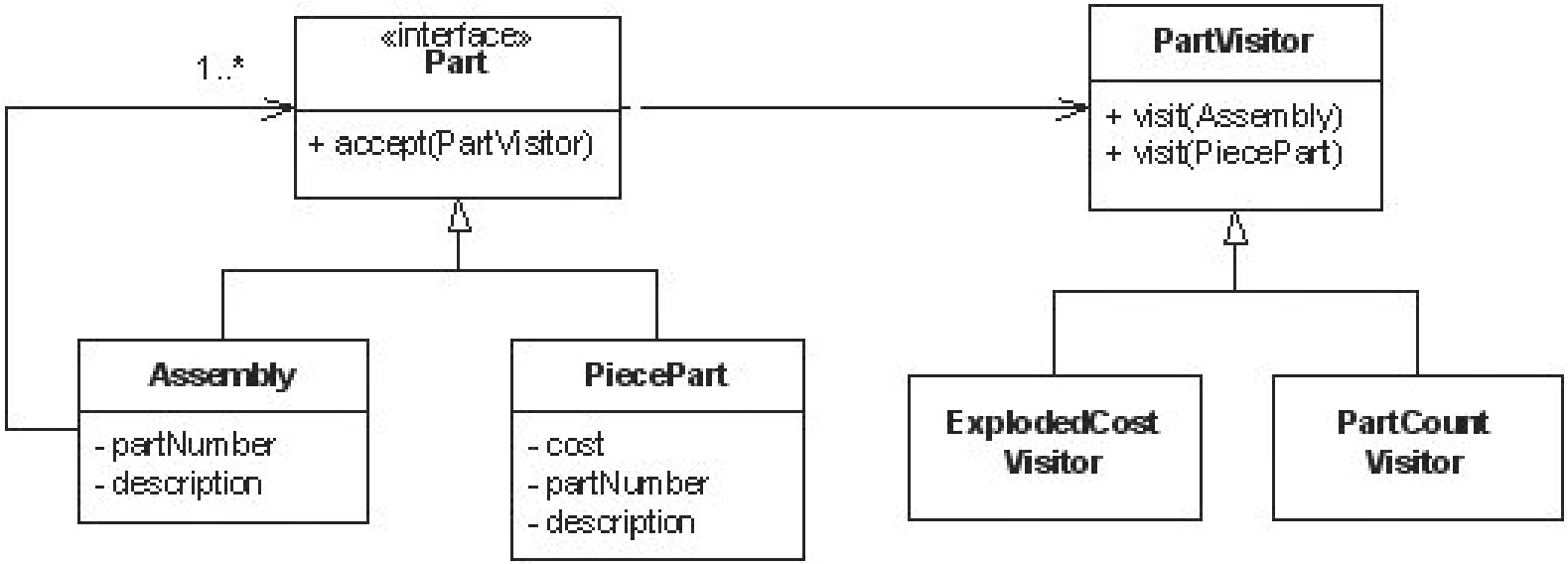
Acyclic Visitor

```
public class UnixModemConfigurator
  implements HayesModemVisitor, ZoomModemVisitor,
     ErnieModemVisitor
{
  public void Visit(HayesModem m)
  {
    m.configurationString = "&s1=4&D=3";
  }

  public void Visit(ZoomModem m)
  {
    m.configurationValue = 42;
  }

  public void Visit(ErnieModem m)
  {
    m.internalPattern = "C is too slow";
  }
}
```

Uses of Visitor

- The VISITOR pattern is commonly used to walk large data structures and to generate reports.
- The value of the VISITOR in this case is that the data structure objects do not have to have any report-generation code.
- New reports can be added by adding new VISITORs rather than by changing the code in the data structures. This means that reports can be placed in separate components and individually deployed only to those customers needing them.
- Consider a simple data structure that represents a bill of materials (BOM).
- We could generate an unlimited number of reports from this data structure. We could generate a report of the total cost of an assembly or a report that listed all the pieceparts in an assembly.

Uses of Visitor



Uses of Visitor

- The Single-Responsibility Principle (SRP) told us that we want to separate code that changes for different reasons.
- The Part hierarchy may change as new kinds of parts are needed.
- However, it should not change because new kinds of reports are needed. Thus, we'd like to separate the reports from the Part hierarchy
- Each new report can be written as a new visitor.
  - We write the Accept function of Assembly to visit the visitor and also call Accept on all the contained Part instances. Thus, the entire tree is traversed. For each node in the tree, the appropriate Visit function is called on the report. The report accumulates the necessary statistics.

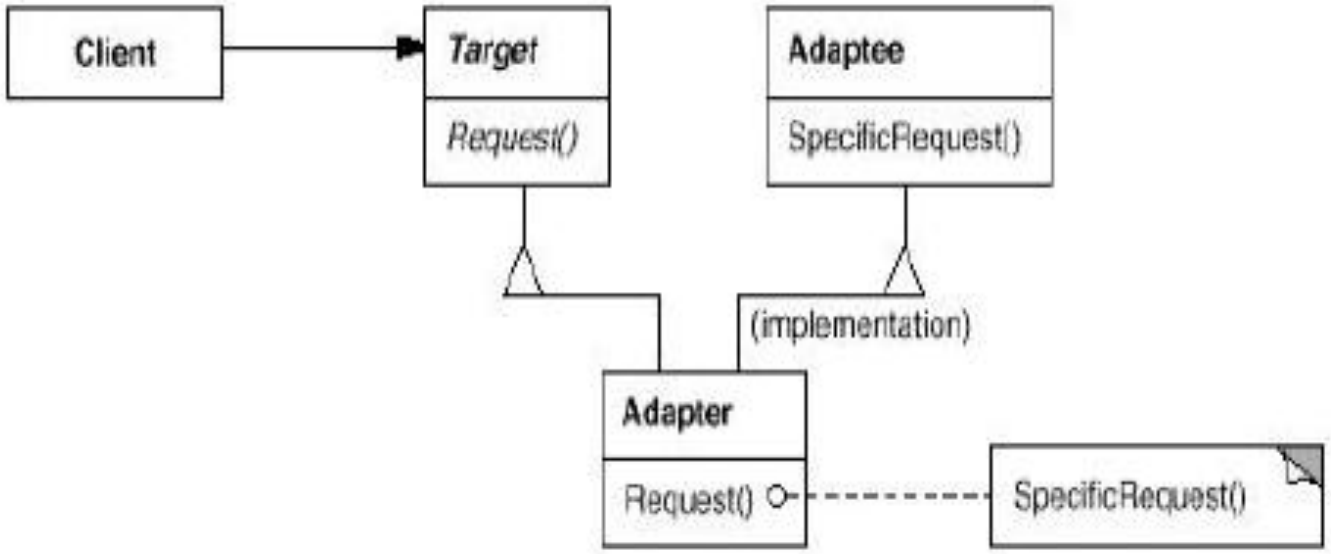
The Adapter Pattern

- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Also Known As
  - Wrapper
- Motivation
  - Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
  - We can not change the library interface, since we may not have its source code
  - Even if we did have the source code, we probably should not change the library for each domain-specific application

The Adapter Pattern

Structure

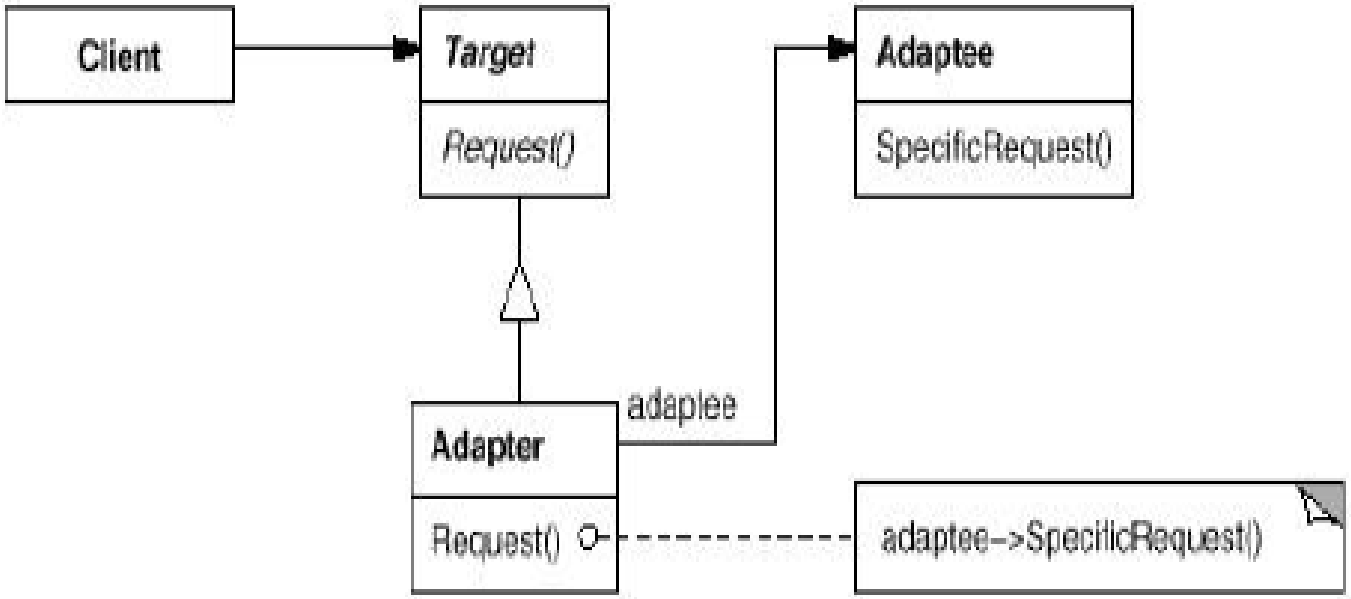
⇒ A class adapter uses multiple inheritance to adapt one interface to another:



The Adapter Pattern

Structure

⇒ An object adapter relies on object composition:

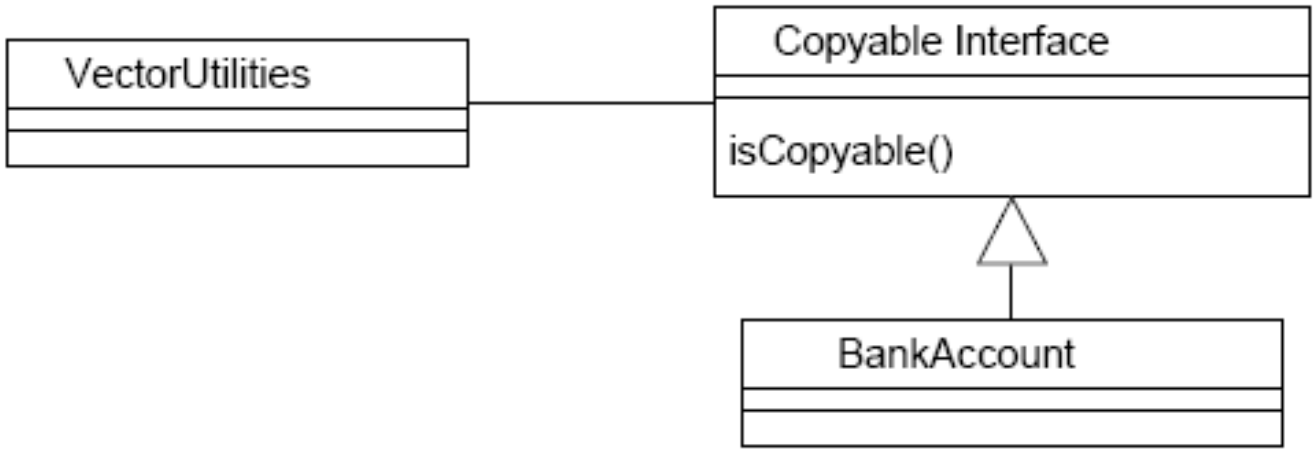


Applicability

- Use the Adapter pattern when
  - You want to use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated classes with incompatible interfaces
- Implementation Issues
  - How much adapting should be done?
    - Simple interface conversion that just changes operation names and order of arguments
    - Totally different set of operations

Example 1

- Consider a utility class that has a copy() method which can make a copy of an vector excluding those objects that meet a certain criteria. To accomplish this the method assumes that all objects in the vector implement the Copyable interface providing the isCopyable() method to determine if the object should be copied or not.



Example 1

- Here's the Copyable interface:

```
public interface Copyable {
  public boolean isCopyable();
}
```

- And here's the copy() method of the VectorUtilities class:

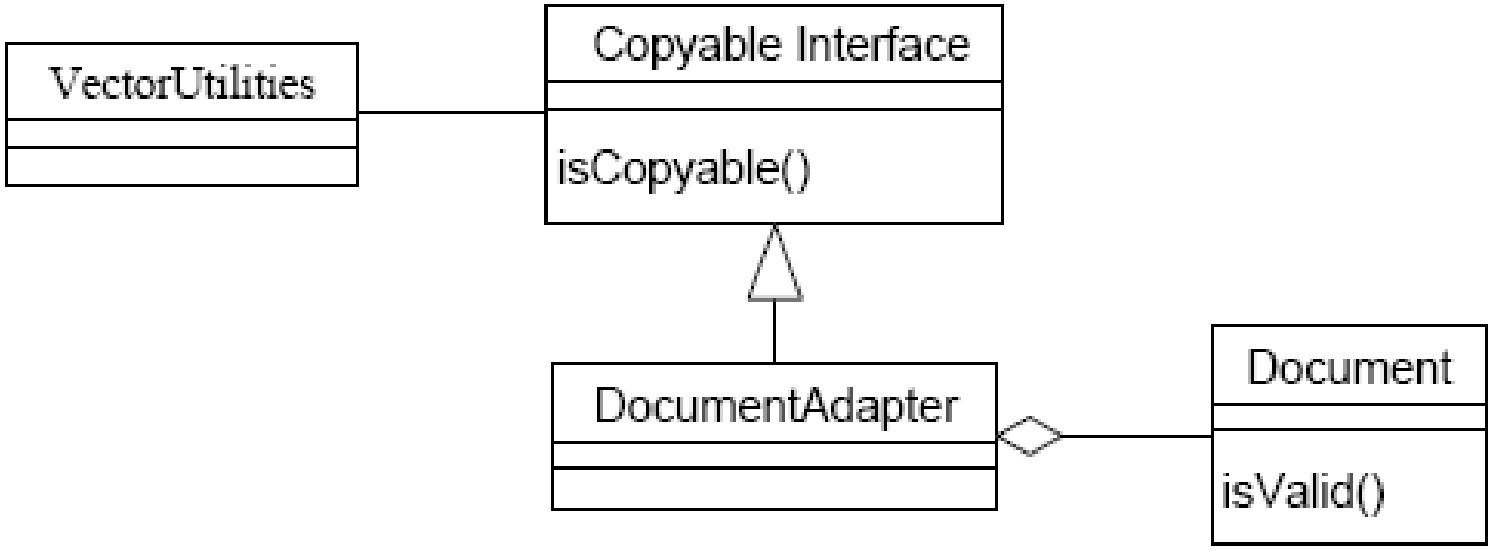
```
public static Vector copy(Vector vin) {
  Vector vout = new Vector();
  Enumeration e = vin.elements();
  while (e.hasMoreElements()) {
    Copyable c = (Copyable) e.nextElement();
    if (c.isCopyable())
      vout.addElemet(c);
  }
  return vout;
}
```

Example 1

- But what if we have a class, say the Document class, that does not implement the Copyable interface. We want to be able perform a selective copy of a vector of Document objects, but we do not want to modify the Document class at all. Sounds like a job for (TA-DA) an adapter!
- To make things simple, let's assume that the Document class has a nice isValid() method we can invoke to determine whether or not it should be copied

Example 1

- Here's our new class diagram:



Example 1

- And here is our DocumentAdapter class:

```
public class DocumentAdapter implements Copyable {

  private Document d;

  public DocumentAdapter(Document d) {
    document = d;
  }

  public boolean isCopyable() {
    return d.isValid();
  }

}
```

Example 2

- Do you see any Adapter pattern here?

```
public class ButtonDemo {

  public ButtonDemo() {
    Button button = new Button("Press me");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         doOperation();
      }
    });
  }
  public void doOperation() { whatever }

}
```

Example 2

- Button objects expect to be able to invoke the actionPerformed() method on their associated ActionListener objects. But the ButtonDemo class does not have this method! It really wants the button to invoke its doOperation() method. The anonymous inner class we instantiated acts as an adapter object, adapting ButtonDemo to ActionListener!

Command

- **Intent**
- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. [GoF, p233]
- Promote "invocation of a method on an object" to full object status
- An object-oriented callback
- **Problem**
- Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

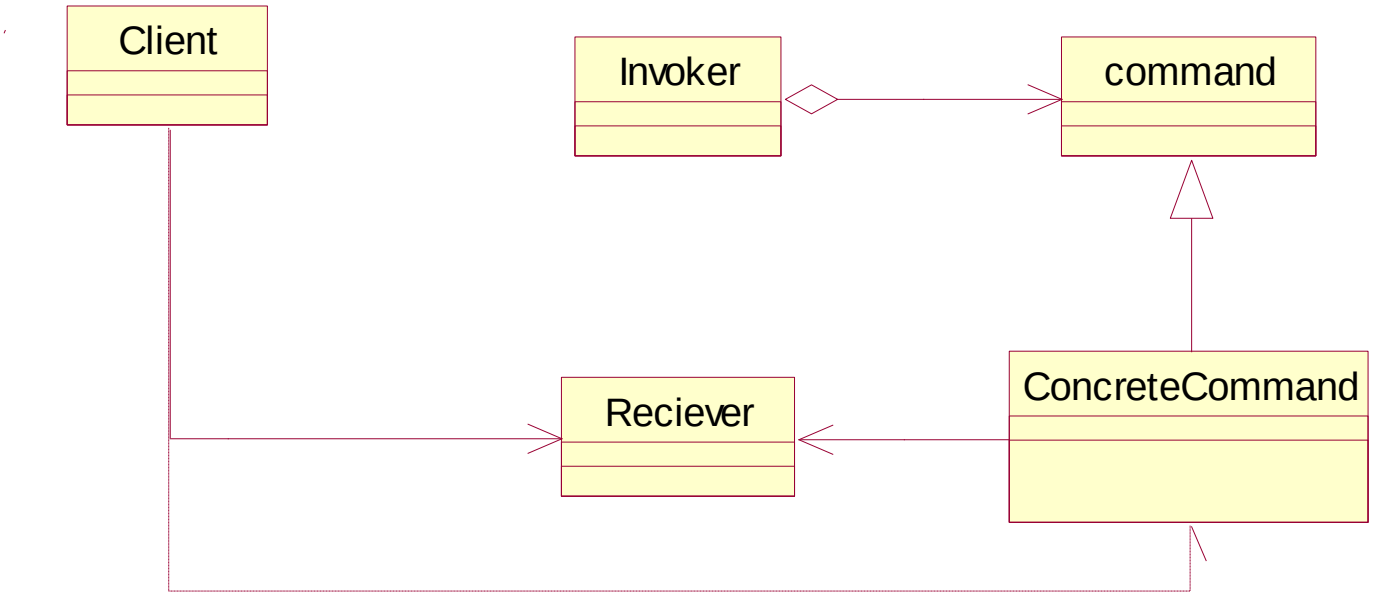
Applicability

This pattern is used when

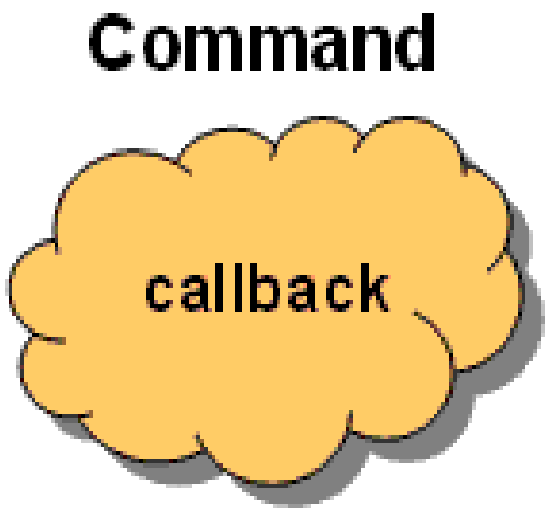
- You have to parameterize objects by an action to perform
- Specify, queue and execute requests at different times.
- Support undo.

Behavioral Patterns – Command

Structure



Command



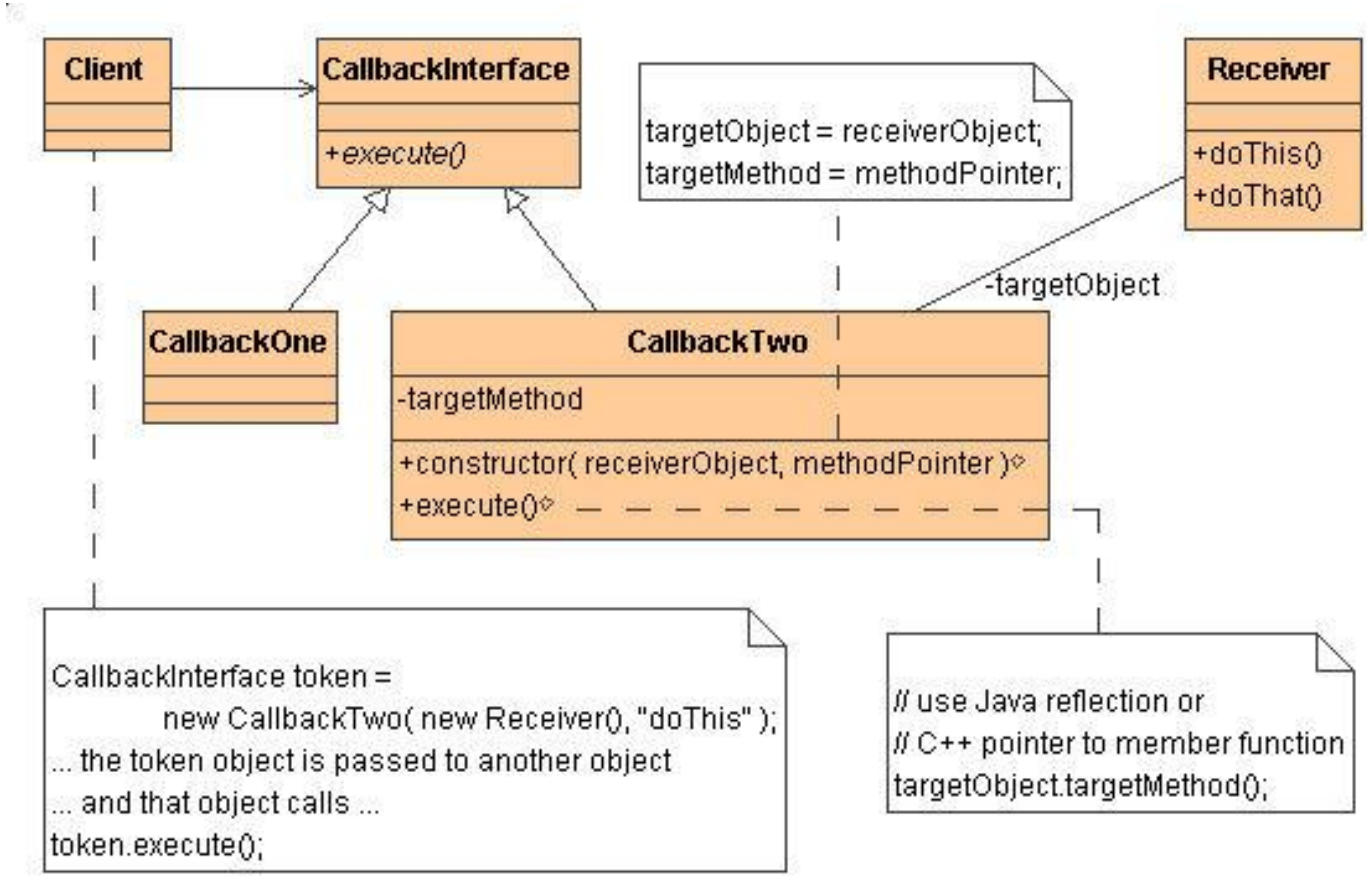
1) Define a class that maintains: a pointer to an object, a pointer to a member function, and all necessary function arguments.

2) Promote "execution of a method on an object" to full object status so that it can be decoupled, deferred, queued, unqueued, archived, or redone.

Command

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an execute() method that simply calls the action on the receiver.
- All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual execute() method whenever the client requires the object's "service".

Command


Client
CallbackInterface
+execute()
targetObject = receiverObject;
targetMethod = methodPointer;
Receiver
+doThis()
+doThat()
-targetObject
CallbackOne
CallbackTwo
-targetMethod
+constructor(receiverObject, methodPointer)
+execute()
CallbackInterface token =
new CallbackTwo(new Receiver(), "doThis");
... the token object is passed to another object
... and that object calls ...
token.execute();
// use Java reflection or
// C++ pointer to member function
targetObject.targetMethod();

Strategy

- "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it." [Gamma, p315]
- Capture the abstraction in an interface, bury implementation details in derived classes.
- Eg: Java's Comparator interface

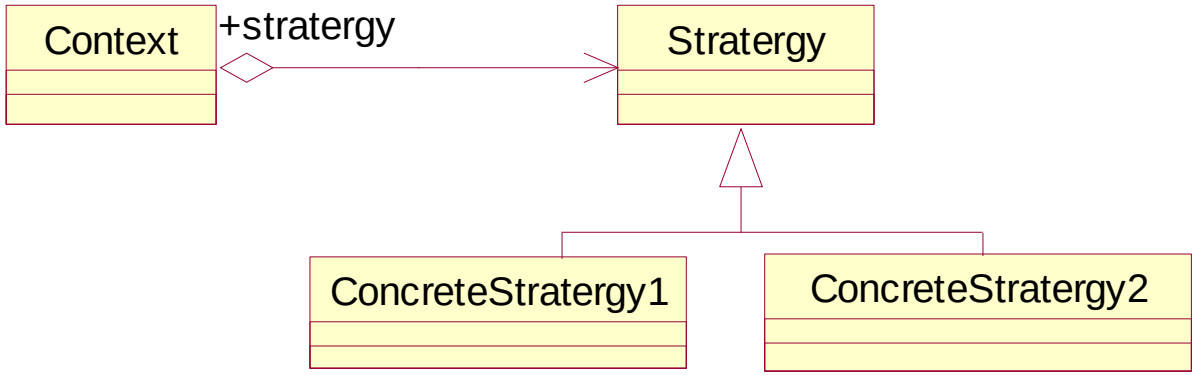
Applicability

This pattern is used when

- Many related classes differ only in their behavior.
- You need different variants of an algorithm

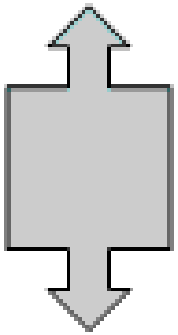
Behavioral Patterns – Strategy

Structure



Strategy

Strategy



standard polymorphism

1) Elevate the interface of an algorithm to an abstract base class.

2) Bury each possible implementation in its own derived class.

3) The client couples herself to the interface, **not** to any particular implementation.

Template Method

- **Intent**
- Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. [GoF, p325]
- Base class declares algorithm “placeholders”, and derived classes implement the placeholders.
- **Problem**
- Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

Behavioral Patterns – Template Method

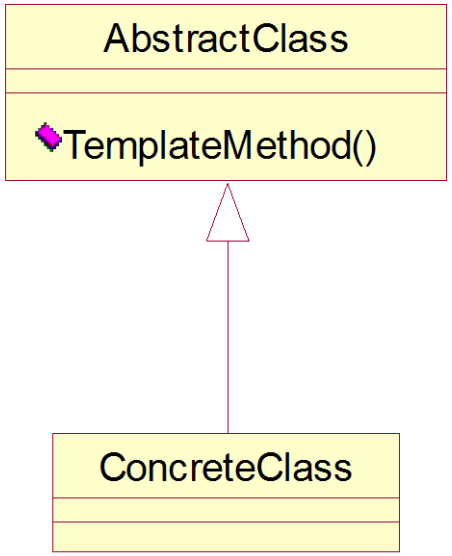
Applicability

This pattern is used when

- You define invariant parts of the algorithm once and let the sub classes implement the behavior that can vary.
- Common behavior among subclasses should be factored and localized in a common class to avoid duplication.
- To control subclass extensions

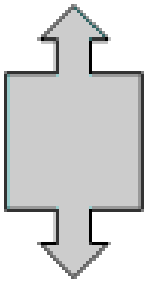
Behavioral Patterns – Template Method

Structure



Template Method

Template Method



define skeleton, defer implementation

1) Evaluate an algorithm with multiple steps, and many possible implementations for one or more of those steps.

2) Where implementation is reusable, define it in a base class.

3) Where multiple implementations are possible: define "place holder" method signatures in the base class, and allocate each possible implementation to its own derived class.

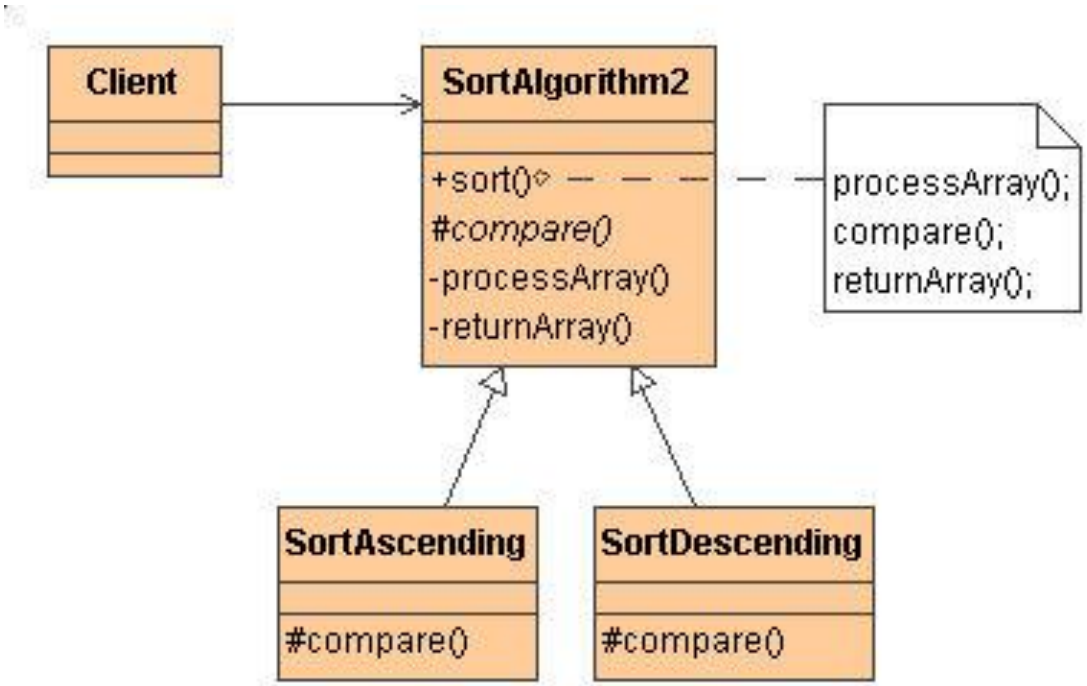
Template Method

- The component designer decides which steps of an algorithm are invariant (or standard), and which are variant (or customizable).
- The invariant steps are implemented in an abstract base class, while the variant steps are either given a default implementation, or no implementation at all. The variant steps represent "hooks", or "placeholders", that can, or must, be supplied by the component's client in a concrete derived class.
- The component designer mandates the required steps of an algorithm, and the ordering of the steps, but allows the component client to extend or replace some number of these steps.

Template Method

- Template Method is used prominently in frameworks.
- Each framework implements the invariant pieces of a domain's architecture, and defines "placeholders" for all necessary or interesting client customization options.
- In so doing, the framework becomes the "center of the universe", and the client customizations are simply "the third rock from the sun".
- This inverted control structure has been affectionately labelled "the Hollywood principle" - "don't call us, we'll call you".

Template Method



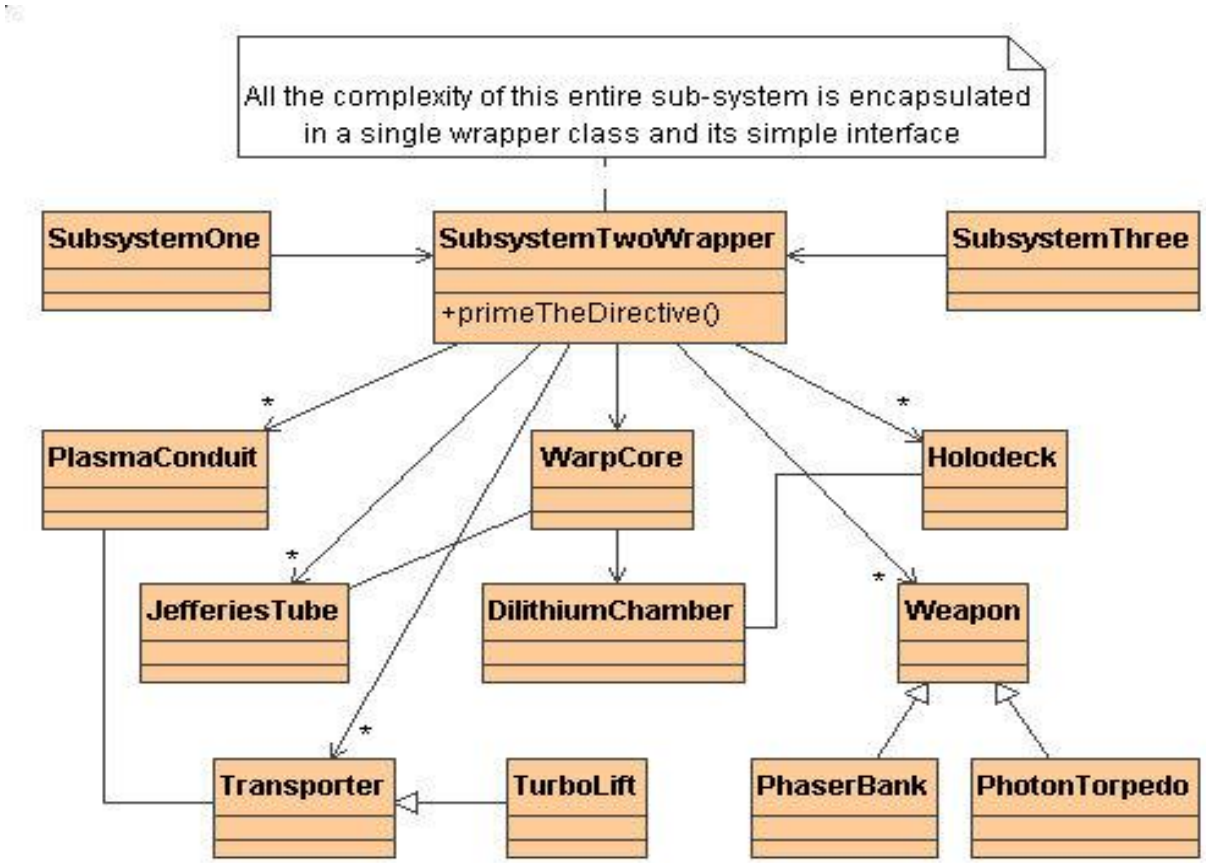
Facade

- **Intent**
- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. [GoF, p185]
- Wrap a complicated subsystem with a simpler interface.
- **Problem**
- A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

Facade

- Facade discusses encapsulating a complex subsystem within a single interface object.
- This reduces the learning curve necessary to successfully leverage the subsystem.
- It also promotes decoupling the subsystem from its potentially many clients.
- On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.
- The Facade object should be a fairly simple advocate or facilitator. It should not become an all-knowing oracle or "god" object.

Facade


All the complexity of this entire sub-system is encapsulated in a single wrapper class and its simple interface
SubsystemOne
SubsystemTwoWrapper
+primeTheDirective()
SubsystemThree
PlasmaConduit
WarpCore
Holodeck
JefferiesTube
DilithiumChamber
Weapon
Transporter
TurboLift
PhaserBank
PhotonTorpedo
*

Observer

The Observer Pattern

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Also Known As
  - Dependents, Publish-Subscribe, Model-View

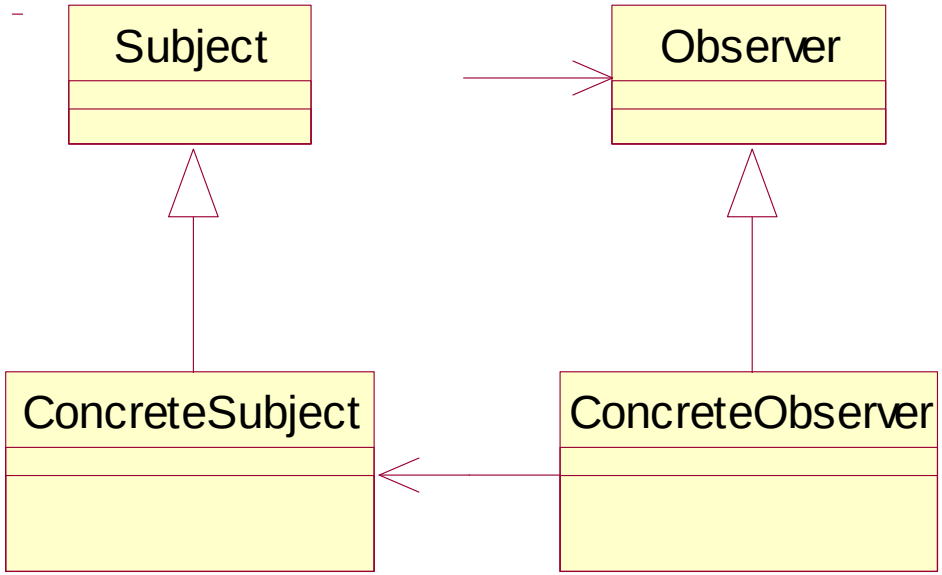
Motivation

- The need to maintain consistency between related objects without making classes tightly coupled

The Observer Pattern

- Applicability
- Use the Observer pattern in any of the following situations:
  - When an abstraction has two aspects, one dependent on the other.
  - Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others
  - When an object should be able to notify other objects without making assumptions about those objects

The Observer Pattern



The Observer Pattern

- Participants
  - Subject
    - Keeps track of its observers
    - Provides an interface for attaching and detaching Observer objects
  - Observer
    - Defines an interface for update notification
  - ConcreteSubject
    - The object being observed
    - Stores state of interest to ConcreteObserver objects
    - Sends a notification to its observers when its state changes
  - ConcreteObserver
    - The observing object
    - Stores state that should stay consistent with the subject's
    - Implements the Observer update interface to keep its state consistent with the subject's

The Observer Pattern

- Consequences
  - Benefits
  - Minimal coupling between the Subject and the Observer
    - Can reuse subjects without reusing their observers and vice versa
    - Observers can be added without modifying the subject
    - All subject knows is its list of observers
    - Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
    - Subject and observer can belong to different abstraction layers
  - Support for event broadcasting
    - Subject sends notification to all subscribed observers
    - Observers can be added/removed at any time

The Observer Pattern

- Implementation Issues
  - How does the subject keep track of its observers?
    - Array, linked list
  - What if an observer wants to observe more than one subject?
    - Have the subject tell the observer who it is via the update interface
  - Who triggers the update?
    - The subject whenever its state changes
    - The observers after they cause one or more state changes
    - Some third party object(s)
  - Make sure the subject updates its state before sending out notifications

Java Implementation Of Observer

- We could implement the Observer pattern “from scratch” in Java
  - But Java provides the Observable/Observer classes as built-in support for the Observer pattern
  - The java.util.Observable class is the base Subject class. Any class that wants to be observed extends this class.
    - Provides methods to add/delete observers
    - Provides methods to notify all observers
    - A subclass only needs to ensure that its observers are notified in the appropriate mutators
    - Uses a Vector for storing the observer references
  - The java.util.Observer interface is the Observer interface. It must be implemented by any observer class.

Observable/Observer Example

```
public class ConcreteSubject extends Observable {
    private String name;
    private float price;
    public ConcreteSubject(String name, float price) {
        this.name = name;
        this.price = price;
        System.out.println("ConcreteSubject created: " + name + " at "+ price);
}
public String getName() {return name;}
public float getPrice() {return price;}
```

Observable/Observer Example

```
public void setName(String name) {
    this.name = name;
    setChanged();
    notifyObservers(name);
}
public void setPrice(float price) {
    this.price = price;
    setChanged();
    notifyObservers(new Float(price));
}
}
```

Observable/Observer Example

```
public class NameObserver implements Observer {
  private String name;

  public NameObserver() {
    name = null;
    System.out.println("NameObserver created: Name is " + name);
  }
  public void update(Observable obj, Object arg) {
    if (arg instanceof String) {
      name = (String)arg;
      System.out.println("NameObserver: Name changed to " + name);
    } else {
      System.out.println("NameObserver: Some other change to
   subject!");
    }
  }
}
```

Observable/Observer Example

```
public class PriceObserver implements Observer {
  private float price;

  public PriceObserver() {
    price = 0;
    System.out.println("PriceObserver created: Price is " + price);
  }
  public void update(Observable obj, Object arg) {
    if (arg instanceof Float) {
      price = ((Float)arg).floatValue();
      System.out.println("PriceObserver: Price changed to " +
                         price);
    } else {
      System.out.println("PriceObserver: Some other change to
                          subject!");
    }
  }
```

Observable/Observer Example

```
// Test program for ConcreteSubject, NameObserver and PriceObserver
public class TestObservers {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
    // Add those Observers!
    s.addObserver(nameObs);
    s.addObserver(priceObs);
    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Sugar Crispies");
  }
}
```

Other examples

- Java AWT/Swing Event model
- JavaBeans/JMX listeners
- JMS

The Chain of Responsibility Pattern

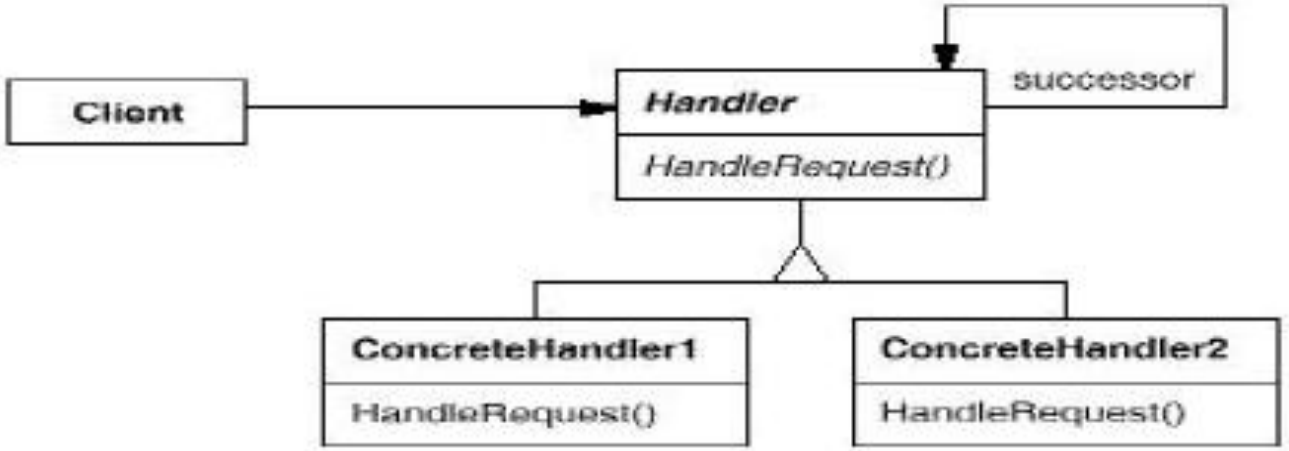
- Intent
  - ⇒ Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Motivation
  - ⇒ Consider a context-sensitive help system for a GUI
  - ⇒ The object that ultimately provides the help isn't known explicitly to the object (e.g., a button) that initiates the help request
  - ⇒ So use a chain of objects to decouple the senders from the receivers. The request gets passed along the chain until one of the objects handles it.
  - ⇒ Each object on the chain shares a common interface for handling requests and for accessing its successor on the chain

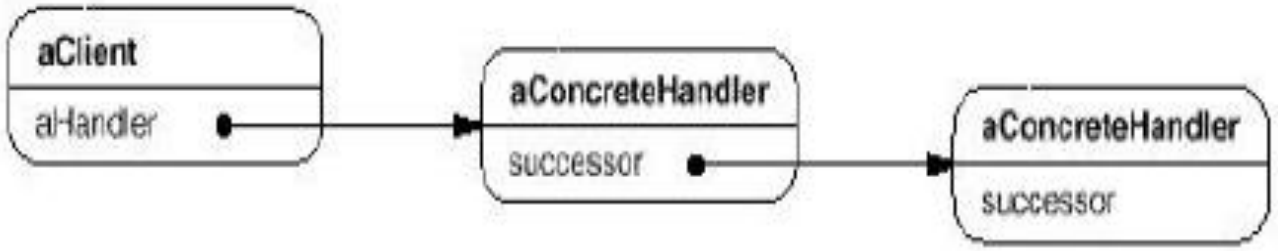
The Chain of Responsibility Pattern

- Applicability
  - ⇒ Use Chain of Responsibility:
    - When more than one object may handle a request and the actual handler is not know in advance
    - When requests follow a "handle or forward" model - that is, some requests can be handled where they are generated while others must be forwarded to another object to be handled
- Consequences
  - ⇒ Reduced coupling between the sender of a request and the receiver - the sender and receiver have no explicit knowledge of each other
  - ⇒ Receipt is not guaranteed - a request could fall off the end of the chain without being handled
  - ⇒ The chain of handlers can be modified dynamically

The Chain of Responsibility Pattern

- Structure





The Chain of Responsibility Pattern

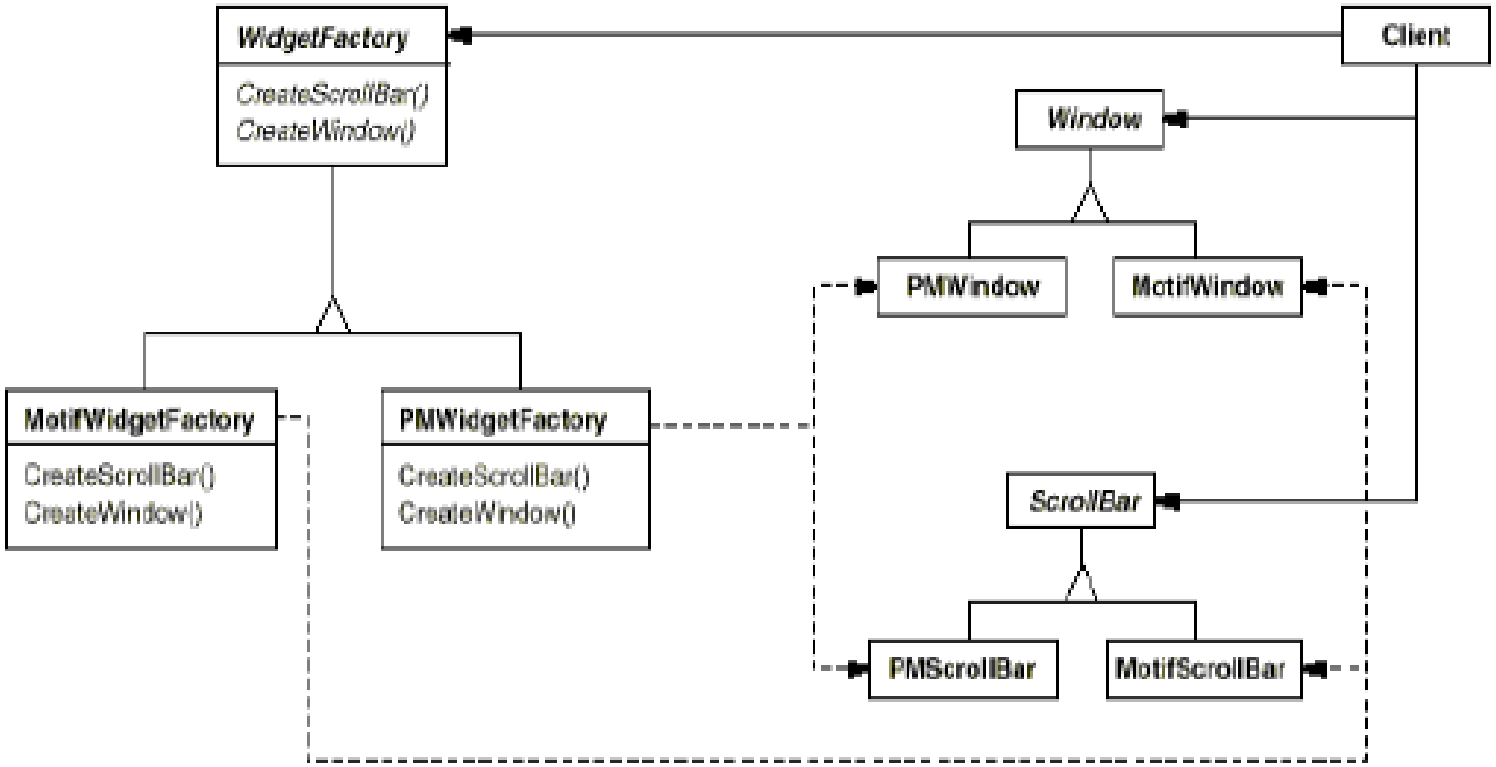
- Scenario: The designers of a set of GUI classes need to have a way to propagate GUI events, such as MOUSE_CLICKED, to the individual component where the event occurred and then to the object or objects that are going to handle the event
- Solution: Use the Chain of Responsibility pattern. First post the event to the component where the event occurred. That component can handle the event or post the event to its container component (or both!). The next component in the chain can again either handle the event or pass it up the component containment hierarchy until the event is handled.
- This technique was actually used in the Java 1.0 AWT

The Abstract Factory Pattern

- Intent
  - ⇒ Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - ⇒ The Abstract Factory pattern is very similar to the Factory Method pattern. One difference between the two is that with the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object via composition whereas the Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.
  - ⇒ Actually, the delegated object frequently uses factory methods to perform the instantiation!

The Abstract Factory Pattern

- Motivation
  - ⇒ A GUI toolkit that supports multiple look-and-feels:



The Abstract Factory Pattern

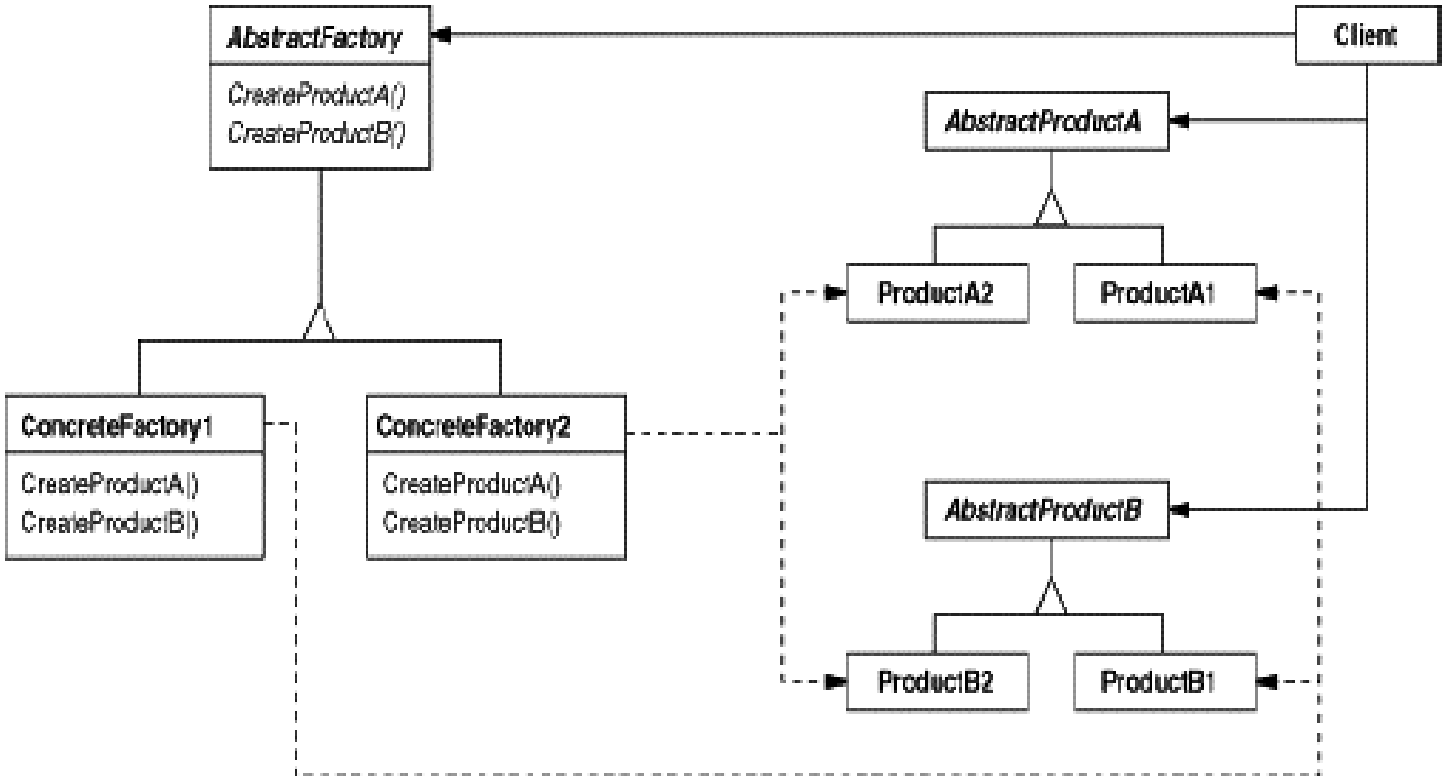
- Applicability

  Use the Abstract Factory pattern in any of the following situations:

  - A system should be independent of how its products are created, composed, and represented
  - A class can't anticipate the class of objects it must create
  - A system must use just one of a set of families of products
  - A family of related product objects is designed to be used together, and you need to enforce this constraint

The Abstract Factory Pattern

- Structure



The Abstract Factory Pattern

- Participants
  - AbstractFactory
    - Declares an interface for operations that create abstract product objects
  - ConcreteFactory
    - Implements the operations to create concrete product objects
  - AbstractProduct
    - Declares an interface for a type of product object
  - ConcreteProduct
    - Defines a product object to be created by the corresponding concrete factory
    - Implements the AbstractProduct interface
  - Client
    - Uses only interfaces declared by AbstractFactory and AbstractProduct classes

The Abstract Factory Pattern

- Collaborations
  - ⇨ Normally a single instance of a ConcreteFactory class is created at run-time. (This is an example of the Singleton Pattern.) This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
  - ⇨ AbstractFactory defers creation of product objects to its ConcreteFactory

Example 1

- Let's see how an Abstract Factory can be applied to the MazeGame
- First, we'll write a MazeFactory class as follows:

```
// MazeFactory.
public class MazeFactory {
  public Maze makeMaze() {return new Maze();}
  public Room makeRoom(int n) {return new Room(n);}
  public Wall makeWall() {return new Wall();}
  public Door makeDoor(Room r1, Room r2) {
    return new Door(r1, r2);}
}
```

- Note that the MazeFactory class is just a collection of factory methods!
- Also, note that MazeFactory acts as both an AbstractFactory and a ConcreteFactory.

Example 1

- Now the createMaze() method of the MazeGame class takes a MazeFactory reference as a parameter:

```
public class MazeGame {

  public Maze createMaze(MazeFactory factory) {
    Maze maze = factory.makeMaze();
    Room r1 = factory.makeRoom(1);
    Room r2 = factory.makeRoom(2);
    Door door = factory.makeDoor(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
    r1.setSide(MazeGame.North, factory.makeWall());
    r1.setSide(MazeGame.East, door);
```

Example 1

```
        r1.setSide(MazeGame.South, factory.makeWall());
        r1.setSide(MazeGame.West, factory.makeWall());
        r2.setSide(MazeGame.North, factory.makeWall());
        r2.setSide(MazeGame.East, factory.makeWall());
        r2.setSide(MazeGame.South, factory.makeWall());
        r2.setSide(MazeGame.West, door);
        return maze;
    }

}
```

- Note how createMaze() delegates the responsibility for creating maze objects to the MazeFactory object

Example 1

- We can easily extend MazeFactory to create other factories:

```
public class EnchantedMazeFactory extends MazeFactory {
  public Room makeRoom(int n) {return new EnchantedRoom(n);}
  public Wall makeWall() {return new EnchantedWall();}
  public Door makeDoor(Room r1, Room r2)
    {return new EnchantedDoor(r1, r2);}
}
```

- In this example, the correlations are:
  - AbstractFactory => MazeFactory
  - ConcreteFactory => EnchantedMazeFactory (MazeFactory is also a ConcreteFactory)
  - AbstractProduct => MapSite
  - ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor
  - Client => MazeGame

Example 2

- The Java 1.1 Abstract Window Toolkit is designed to provide a GUI interface in a heterogeneous environment
- The AWT uses an Abstract Factory to generate all of the required peer components for the specific platform being used
- For example, here's part of the List class:

```
public class List extends Component implements ItemSelectable {
...
  peer = getToolkit().createList(this);
...
}
```

- The getToolkit() method is inherited from Component and returns a reference to the factory object used to create all AWT widgets

Example 2

- Here's the getToolkit() method in Component:

```
public Toolkit getToolkit() {
  // If we already have a peer, return its Toolkit.
  ComponentPeer peer = this.peer;
  if ((peer != null) && ! (peer instanceof
     java.awt.peer.LightweightPeer)){
      return peer.getToolkit();
  }
  // If we are already in a container, return its Toolkit.
  Container parent = this.parent;
  if (parent != null) {
    return parent.getToolkit();
  }
  // Else return the default Toolkit.
  return Toolkit.getDefaultToolkit();
}
```

Example 2

- And here's the getDefaultToolkit() method in Toolkit:

```
public static synchronized Toolkit getDefaultToolkit() {
  if (toolkit == null)
    String nm = System.getProperty("awt.toolkit",
                                   "sun.awt.motif.MToolkit");
    toolkit = (Toolkit)Class.forName(nm).newInstance();
  }
  return toolkit;
}
```

The Abstract Factory Pattern

- Consequences
  - ⇨ Benefits
    - Isolates clients from concrete implementation classes
    - Makes exchanging product families easy, since a particular concrete factory can support a complete family of products
    - Enforces the use of products only from one family
  - ⇨ Liabilities
    - Supporting new kinds of products requires changing the AbstractFactory interface
- Implementation Issues
  - ⇨ How many instances of a particular concrete factory should there be?
    - An application typically only needs a single instance of a particular concrete factory
    - Use the Singleton pattern for this purpose

The Abstract Factory Pattern

- Implementation Issues
  - How can the factories create the products?
    - Factory Methods
    - Factories
  - How can new products be added to the AbstractFactory interface?
    - AbstractFactory defines a different method for the creation of each product it can produce
    - We could change the interface to support only a make(String kindOfProduct) method

The Singleton Pattern

Sometimes it is appropriate to have exactly one instance of a class:

- window managers,
- print spoolers,
- filesystems.

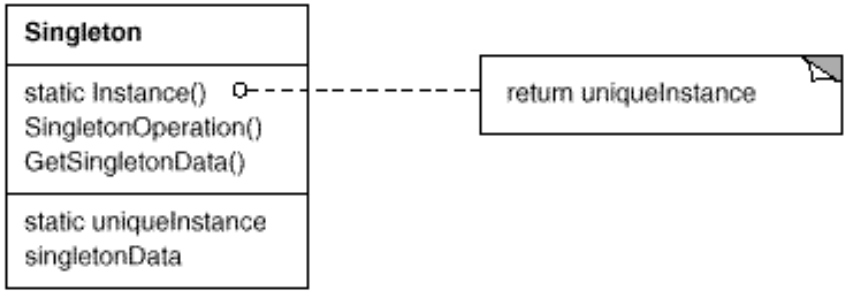
Typically, those types of objects known as singletons, are accessed by disparate objects throughout a software system, and therefore require a global point of access.

The Singleton pattern addresses all the concerns above. With the Singleton design pattern you can:

- Ensure that only one instance of a class is created.
- Provide a global point of access to the object.
- Allow multiple instances in the future without affecting a singleton class' clients.

The Singleton Pattern

- The Singleton pattern ensures a class has only one instance, and provides a global point of access to it.
- The class itself is responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (**by intercepting requests to create new objects**), and it can provide a way to access the instance.
- Singletons maintain a static reference to the **sole singleton instance** and return a reference to that instance from a static *instance()* method.



The Singleton Pattern

```
public class ClassicSingleton {
    private static ClassicSingleton instance = null;

    protected ClassicSingleton() {
            // exists only to defeat instantiation.
    }
    public static ClassicSingleton getInstance() {
        if(instance == null) {
            instance = new ClassicSingleton();
        }
        return instance;
    }
}
```

The *ClassicSingleton* class maintains a static reference to the lone singleton instance and returns that reference from the static getInstance() method.

The Singleton Pattern

- The ClassicSingleton class employs a technique known as lazy instantiation to create the singleton; as a result, the singleton instance is not created until the getInstance() method is called for the first time. This technique ensures that singleton instances are created only when needed.
- The ClassicSingleton class implements a protected constructor so clients cannot instantiate ClassicSingleton instances; however, the following code is perfectly legal:

```
public class SingletonInstantiator {
    public SingletonInstantiator() {
        ClassicSingleton instance =
ClassicSingleton.getInstance();

        ClassicSingleton anotherInstance = new
ClassicSingleton();      ...   } }
```

The Singleton Pattern

How can a class that does not extend *ClassicSingleton* create a *ClassicSingleton* instance if the *ClassicSingleton* constructor is protected?

- Protected constructors can be called by subclasses and *by other classes in the same package*. Hence, because *ClassicSingleton* and *SingletonInstantiator* are in the same package (the default package), *SingletonInstantiator()* methods can create *ClassicSingleton* instances.

Solutions:

1. We can make the ***ClassicSingleton* constructor private** so that only ClassicSingleton's methods call it; however, that means *ClassicSingleton* cannot be subclassed. Also, it's a good idea to declare the singleton class **final**, which makes that intention explicit and allows the compiler to apply performance optimizations.
2. We can put your singleton class in an explicit package, so classes in other packages (including the default package) cannot instantiate singleton instances.

The Singleton Pattern

The ClassicSingleton class is not thread-safe.

If two threads – we will call them Thread 1 and Thread 2, call *ClassicSingleton.getInstance()* at the same time, two *ClassicSingleton* instances can be created if Thread 1 is preempted just after it enters the if block and control is subsequently given to Thread 2.

Solution: **Synchronization**

```
public class ClassicSingleton {
    private static ClassicSingleton instance = null;
    private static Object syncObject; // needed to synchronize  a block
    protected ClassicSingleton() {/*exists only to defeat instantiation*/};
    public static ClassicSingleton getInstance() {
        synchronized(syncObject) {
                if (instance == null) instance = new ClassicSingleton();
        }
        return instance;
    }
}
```

The Singleton Pattern

- It can be difficult to subclass a Singleton, since this can only work if the base Singleton class has not yet been instantiated.
- We can easily change a Singleton to allow a small number of instances where this is allowable and meaningful.
- We can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
- The Singleton pattern permits refinement of operations and representation. The Singleton class may be subclassed, and it is easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.