哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 6: Maintainability-Oriented Software Construction Approaches

6.2 Design Patterns for Maintainability

Ming Liu

April 8, 2019

Outline

- **Creational patterns**
  - Factory method pattern creates objects without specifying the exact class to create.
  - Abstract factory pattern groups object factories that have a common theme.
  - Builder pattern constructs complex objects by separating construction and representation.
- **Structural patterns**
  - Bridge decouples an abstraction from its implementation so that the two can vary independently.
  - Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.
  - Composite composes zero-or-more similar objects so that they can be manipulated as one object.

Outline

- **Behavioral patterns（*）**
  - Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
  - Observer is a publish/subscribe pattern which allows a number of observer objects to see an event.
  - Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object.
  - Chain of responsibility delegates commands to a chain of processing objects.
  - Command creates objects which encapsulate actions and parameters.

Design patterns taxonomy

- **Creational patterns**
  - Concern the process of object creation
- **Structural patterns**
  - Deal with the composition of classes or objects
- **Behavioral patterns**
  - Characterize the ways in which classes or objects interact and distribute responsibility.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

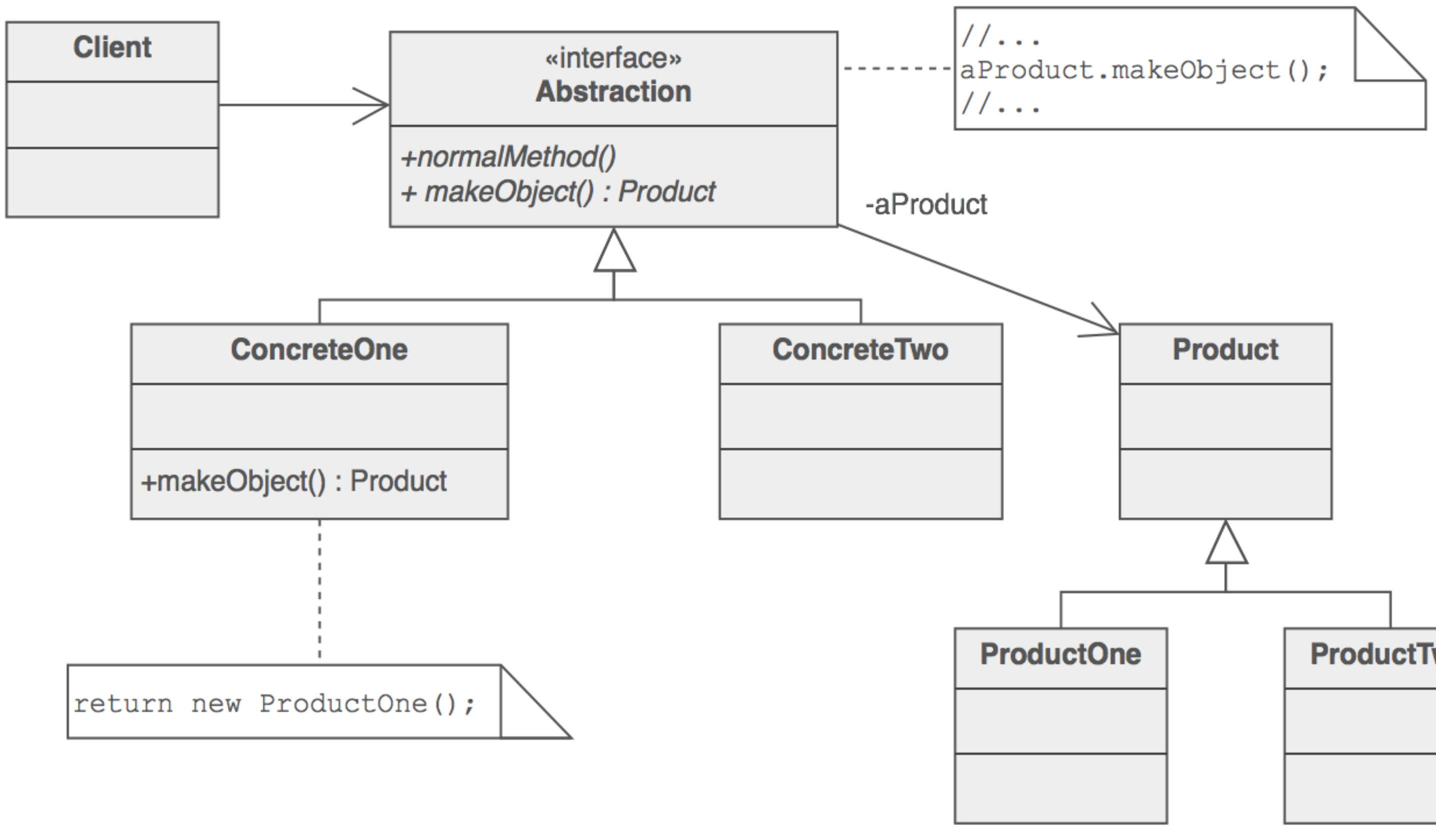
1 Creational patterns

(1) Factory Method pattern

Factory Method

- **Also known as "Virtual Constructor"**
- **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.
- **When should we use Factory Method? ---- When a class:**
  - Can't predict the class of the objects it needs to create
  - Wants its subclasses to specify the objects that it creates
  - Delegates responsibility to one of multiple helper subclasses, and you need to localize the knowledge of which helper is the delegate.

Factory Method



Example

Abstract product

```
public interface Trace {
      // turn on and off debugging
      public void setDebug( boolean debug );
      // write out a debug message
      public void debug( String message );
      // write out an error message
      public void error( String message );
}
```

Concrete product 1

```
public class FileTrace implements Trace {
      private PrintWriter pw;
      private boolean debug;
      public FileTrace() throws IOException {
         pw = new PrintWriter( new FileWriter( "t.log" ) );
      }
      public void setDebug( boolean debug ) {
         this.debug = debug;
      }
      public void debug( String message ) {
        if( debug ) {
            pw.println( "DEBUG: " + message );
            pw.flush();
        }
      }
      public void error( String message ) {
        pw.println( "ERROR: " + message );
        pw.flush();
      }
}
```

Example

Abstract product

```
public interface Trace {
      // turn on and off debugging
      public void setDebug( boolean debug );
      // write out a debug message
      public void debug( String message );
      // write out an error message
      public void error( String message );
}
```

Concrete product 2

```
public class SystemTrace implements Trace {
      private boolean debug;
      public void setDebug( boolean debug ) {
        this.debug = debug;
      }
      public void debug( String message ) {
        if( debug )
           System.out.println( "DEBUG: " + message );
      }
      public void error( String message ) {
        System.out.println( "ERROR: " + message );
      }
}
```

How to use?

```
//... some code ...
SystemTrace log = new SystemTrace();
log.debug( "entering log" );

FileTrace log2 = new FileTrace();
log.debug("...");
```

The client code is tightly coupled with concrete products.

Example

```
public class TraceFactory1 {
      public static Trace getTrace() {
            return new SystemTrace();
      }
}

public class TraceFactory2 {
      public static Trace getTrace(String type) {
         if(type.equals(“file”)
             return new FileTrace();
         else if (type.equals(“system”)
             return new SystemTrace();
      }
}
```

```
//... some code ...
Trace log = TraceFactory1.getTrace();
log.debug( "entering log" );

Trace log = TraceFactory2.getTrace(“system”);
log.debug(“...”);
```

Factory Method

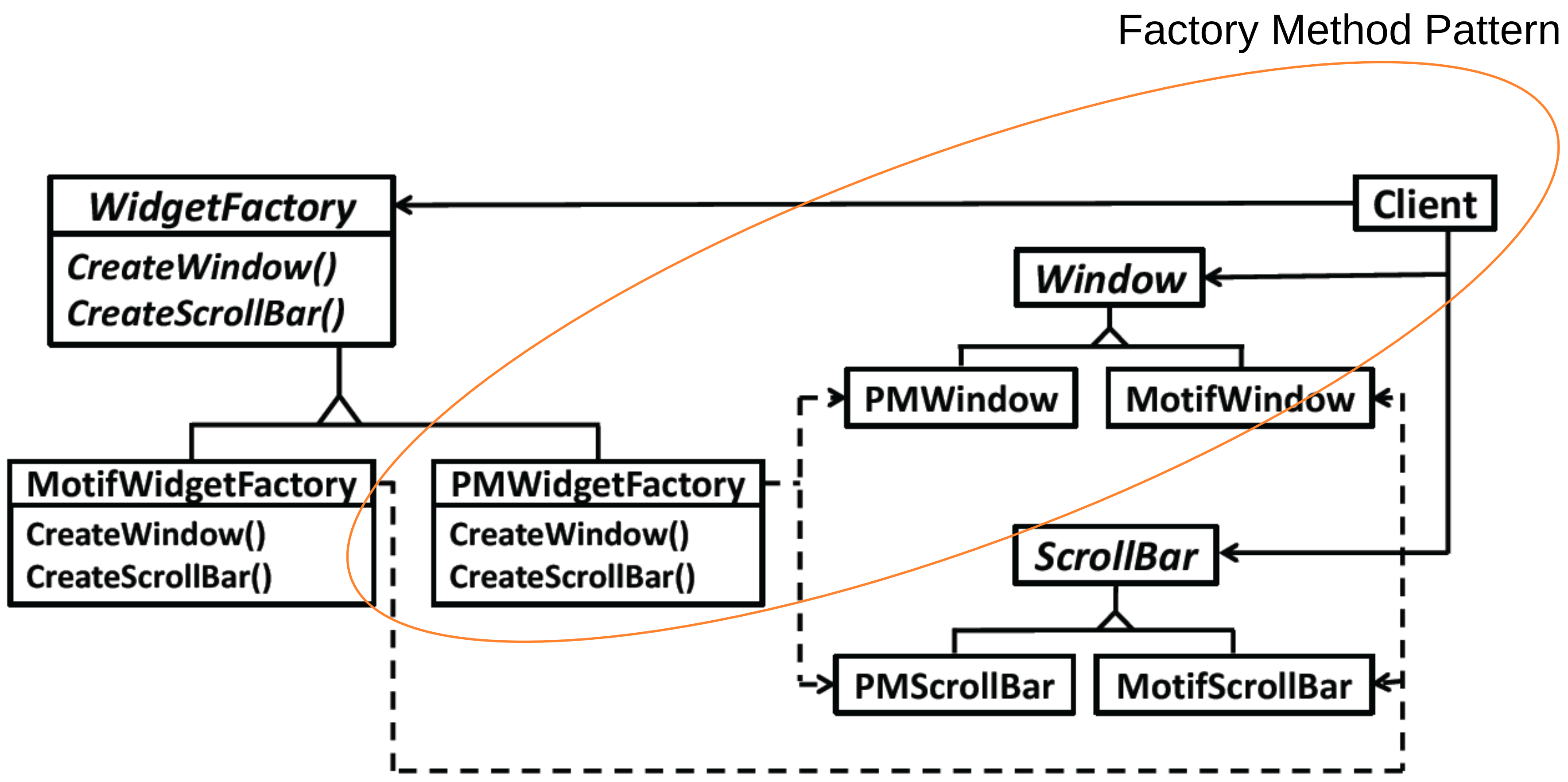
- **Advantage:**
  - Elminates the need to bind application-specific classes to your code.
  - Code deals only with the Product interface (`Trace`), so it can work with any user-defined ConcreteProduct (`FileTrace`, `SystemTrace`)
- **Potential Disadvantages**
  - Clients may have to make a subclass of the Creator, just so they can create a certain ConcreteProduct.
  - This would be acceptable if the client has to subclass the Creator anyway, but if not then the client has to deal with another point of evolution.

(2) Abstract Factory

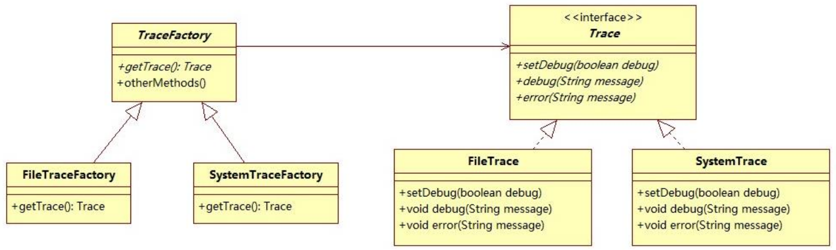
Abstract Factory Pattern Motivation

- **Consider a user interface toolkit that supports multiple looks and feel standards for different operating systems:**
  - How can you write a single user interface and make it portable across the different look and feel standards for these window managers?
- **Consider a facility management system for an intelligent house that supports different control systems:**
  - How can you write a single control system that is independent from the manufacturer?

Structure of Abstract Factory



Example



```
public abstract class TraceFactory {
      public abstract Trace getTrace();
      //other methods…
}



public class SystemTraceFactory extends TraceFactory {
      public Trace getTrace() {
            return new SystemTrace();
      }
}

public class FileTraceFactory extends TraceFactory {
      public Trace getTrace() {
            return new FileTrace();
      }
}
```

```
//... some code ...
TraceFactory traceFactory = new SystemTraceFactory();
Trace log = traceFactory.getTrace();
log.debug( "entering log" );

traceFactory = new FileTraceFactory();
Trace log = TraceFactory.getTrace();
log.debug(“...”);
```

Example --static factory method

```
public class TraceFactory1 {
      public static Trace getTrace() {
            return new SystemTrace();
      }
}

public class TraceFactory2 {
      public static Trace getTrace(String type) {
         if(type.equals(“file”)
             return new FileTrace();
         else if (type.equals(“system”)
             return new SystemTrace();
      }
}
```

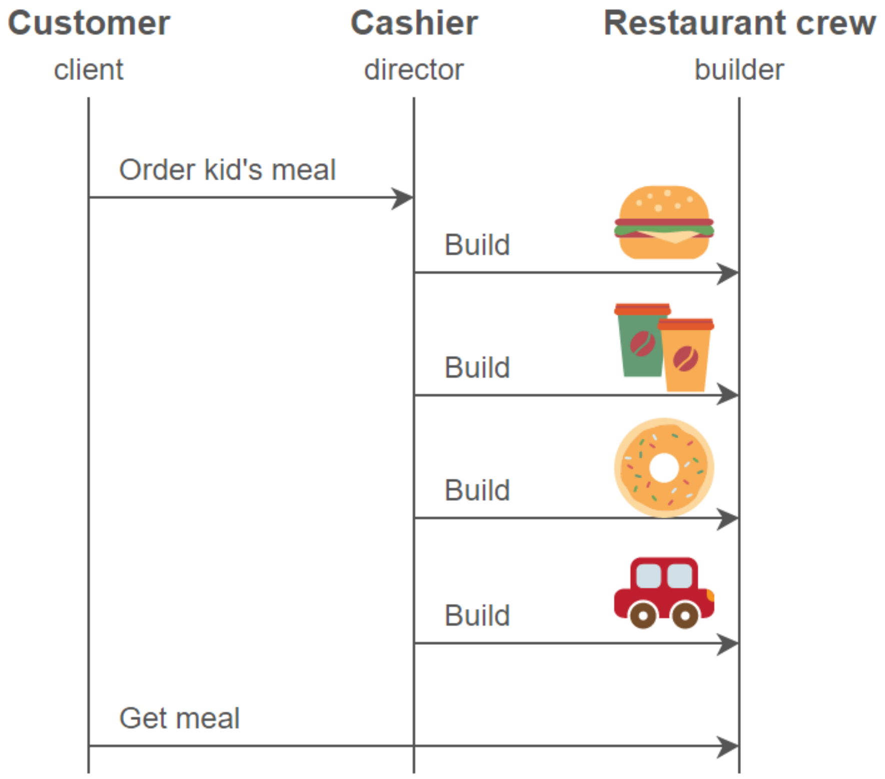
```
//... some code ...
Trace log = TraceFactory1.getTrace();
log.debug( "entering log" );

Trace log = TraceFactory2.getTrace(“system”);
log.debug(“...”);
```

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(3) Builder

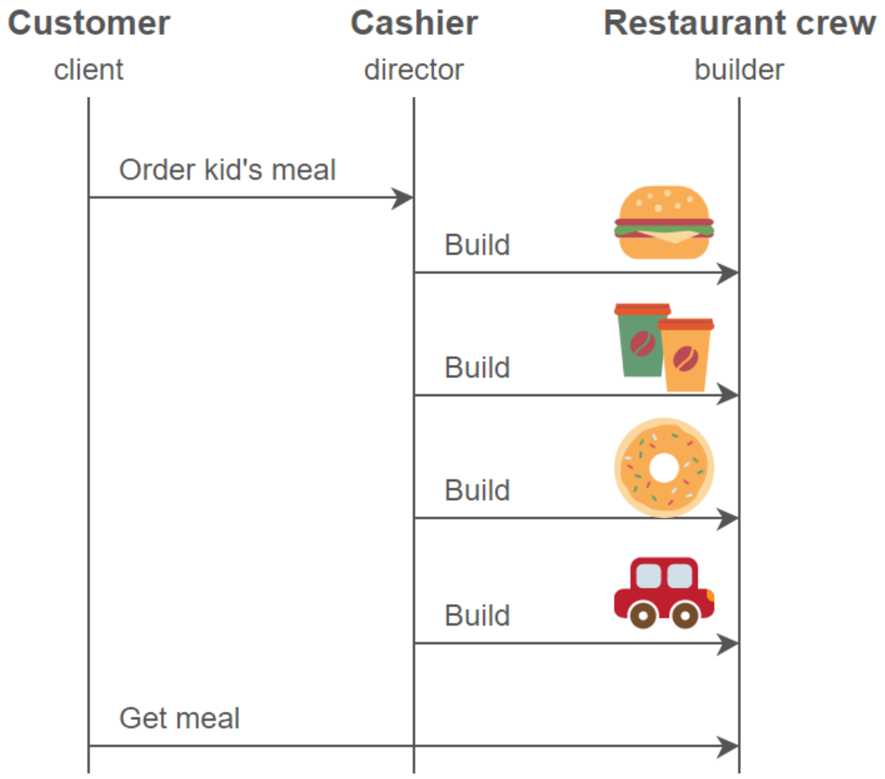
Builder Pattern

- **Builder Pattern:** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - The construction of a complex object is common across several representations
- **Example: Converting a document to a number of different formats**
  - The steps for writing out a document are the same
  - The specifics of each step depend on the format
- **Just like you order a combo food in McDonalds!**

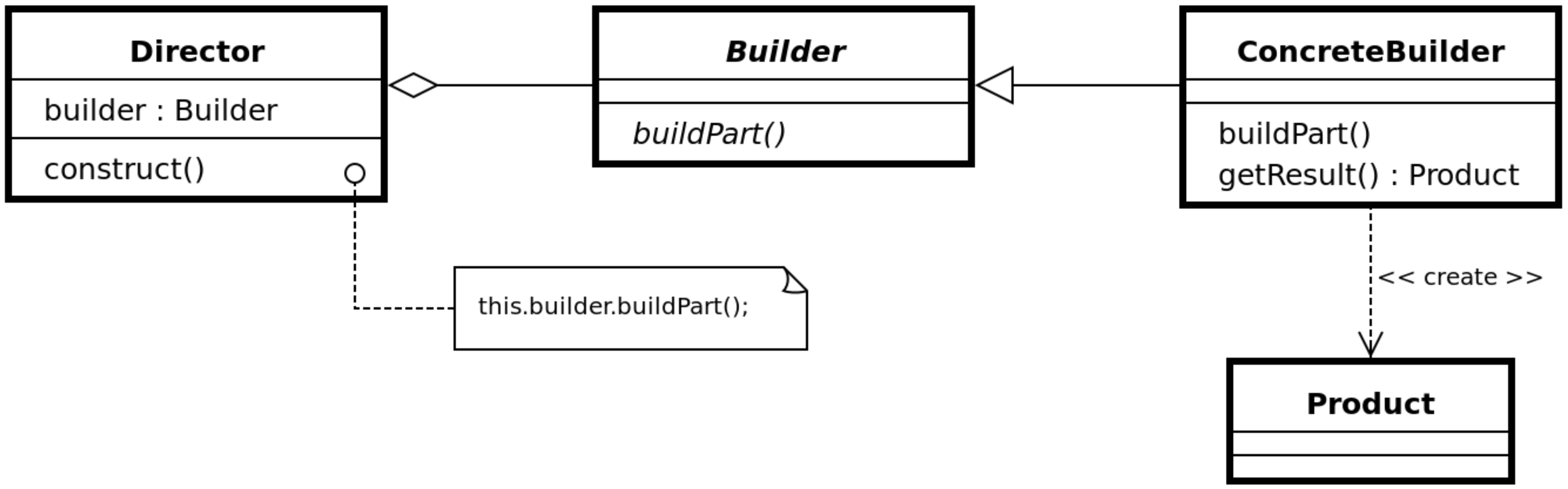


Builder Pattern

- **Approach**
  - The construction algorithm is specified by a single class (the "director")
  - The abstract steps of the algorithm (one for each part) are specified by an interface (the "builder")
  - Each representation provides a concrete implementation of the interface (the "concrete builders")



Builder Pattern



Example

```
/* "Product" */
class Pizza {
    private String dough = "";
    private String sauce = "";
    private String topping = "";

    public void setDough(String dough) {
        this.dough = dough;
    }
    public void setSauce(String sauce) {
        this.sauce = sauce;
    }
    public void setTopping(String topping) {
        this.topping = topping;
    }
}
```

```
/* "Abstract Builder" */
abstract class PizzaBuilder {
    protected Pizza pizza;
    public Pizza getPizza() {
        return pizza;
    }
    public void createNewPizzaProduct() {
        pizza = new Pizza();
    }
    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
}
```

Example

```java
/* "ConcreteBuilder 1" */
class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough() {
        pizza.setDough("pan baked");
    }
    public void buildSauce() {
        pizza.setSauce("hot");
    }
    public void buildTopping() {
        pizza.setTopping("pepperoni+salami");
    }
}
```

```java
/* "ConcreteBuilder 2" */
class HawaiianPizzaBuilder extends
PizzaBuilder {
    public void buildDough() {
        pizza.setDough("cross");
    }
    public void buildSauce() {
        pizza.setSauce("mild");
    }
    public void buildTopping() {
        pizza.setTopping("ham+pineapple");
    }
}
```

Example

```java
/* "Director" */
class Waiter {
    private PizzaBuilder pizzaBuilder;
    public void setPizzaBuilder(PizzaBuilder pb) {
        pizzaBuilder = pb;
    }
    public Pizza getPizza() {
        return pizzaBuilder.getPizza();
    }
    public void constructPizza() {
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}
```

```java
/* A customer ordering a pizza. */
public class PizzaBuilderDemo {
    public static void main(String[] args) {
        Waiter waiter = new Waiter();
        PizzaBuilder hawaiianPizzabuilder = new HawaiianPizzaBuilder();
        PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

        waiter.setPizzaBuilder( hawaiianPizzabuilder );
        waiter.constructPizza();
        Pizza pizza = waiter.getPizza();
    }
}
```

Comparison: Abstract Factory vs Builder

- **Abstract Factory**
  - Focuses on product family (similar product types)
  - Does not hide the creation process
- **Builder**
  - The underlying product needs to be constructed as part of the system, but the creation is very complex (composition of products)
  - The construction of the complex product changes from time to time
  - Hides the creation process from the user
- **Abstract Factory and Builder work well together for a family of multiple complex products**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

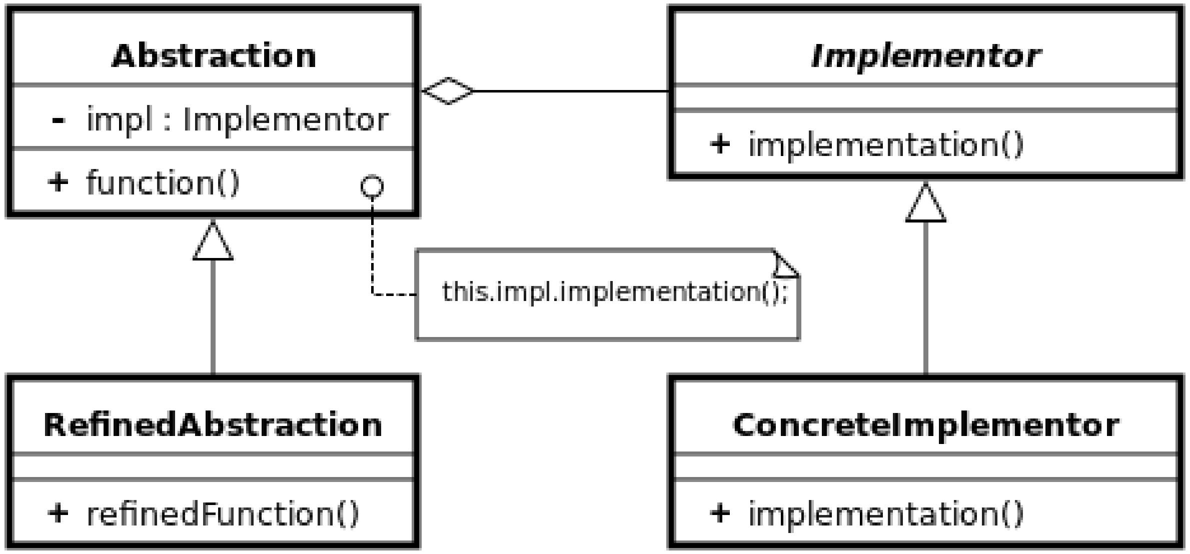
2 Structural patterns

(1) Bridge

Bridge Pattern

- **Applicability**
  - Decouple abstract concept with different implementations
  - Implementation may be switched at run-time
  - Implementation changes should not affect clients
  - Hide a class's interface from clients
- **Structure: use two hierarchies**
  - Logical one for clients
  - Physical one for different implementations
- **Object: to improve extensibility**
  - Logical classes and physical classes change independently
  - Hides implementation details from clients

Structure of Bridge Pattern



Example

Create bridge implementer interface

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}
```

Create concrete bridge implementer classes

```
public class DrawRedCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Color: red " + radius + x + y);
   }
}
public class DrawGreenCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Color: green " + radius + x + y);
   }
}
```

Example

Create an abstract class Shape using the `DrawAPI` interface

```
public abstract class Shape {
   protected DrawAPI drawAPI;
   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw();
}
```

Create concrete class implementing the `Shape` interface

```
public class Circle extends Shape {
   private int x, y, radius;

   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }
   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
```

Example

Use the `Shape` and `DrawAPI` classes to draw different colored circles

```
public class BridgePatternDemo {

   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100, 10, new DrawRedCircle());
      Shape greenCircle = new Circle(100,100, 10, new DrawGreenCircle());

      redCircle.draw();
      greenCircle.draw();
   }
}
```

(2) Proxy

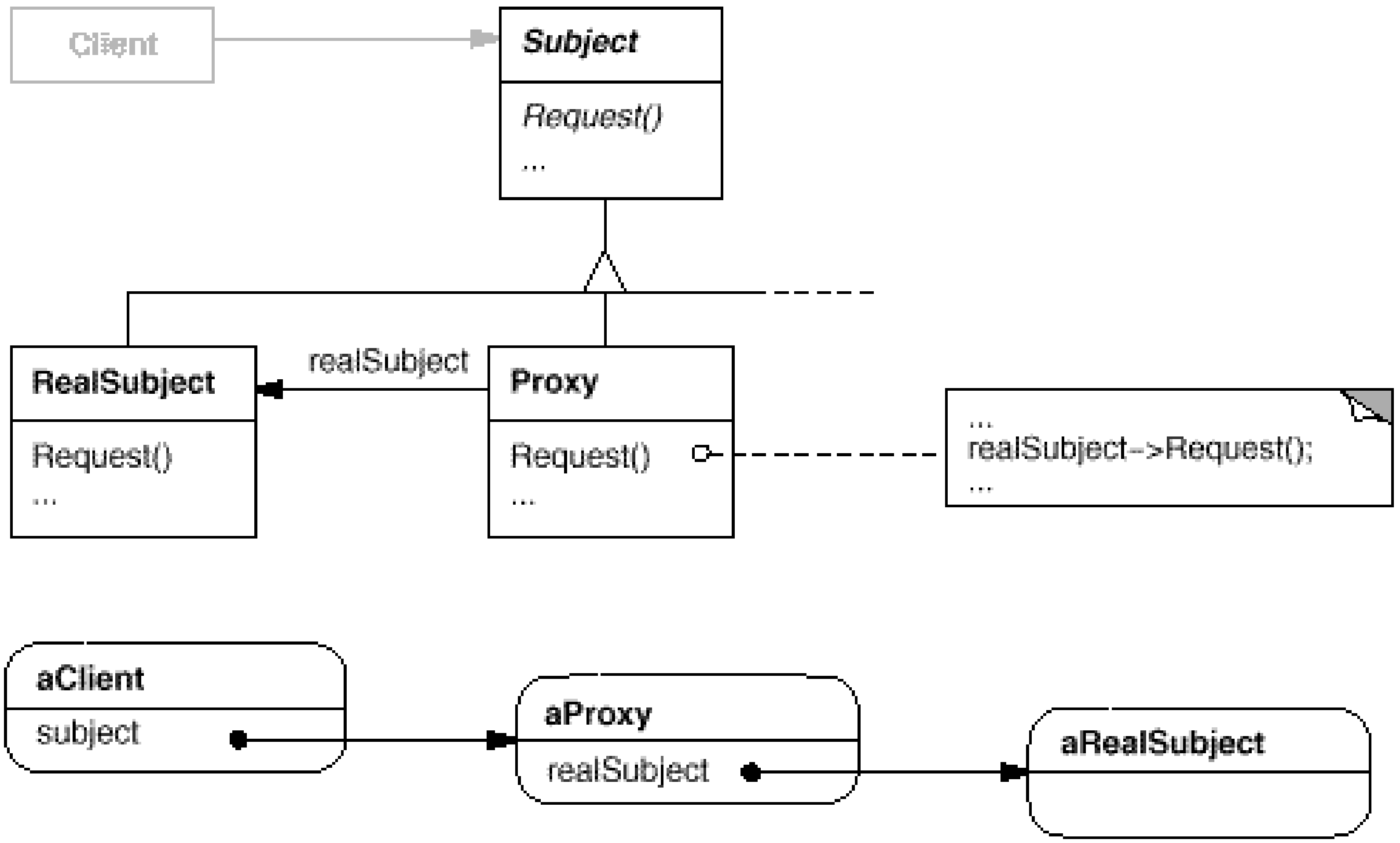
Proxy Pattern Motivation

- **Goal:**
  - Prevent an object from being accessed directly by its clients
- **Solution:**
  - Use an additional object, called a proxy
  - Clients access to protected object only through proxy
  - Proxy keeps track of status and/or location of protected object

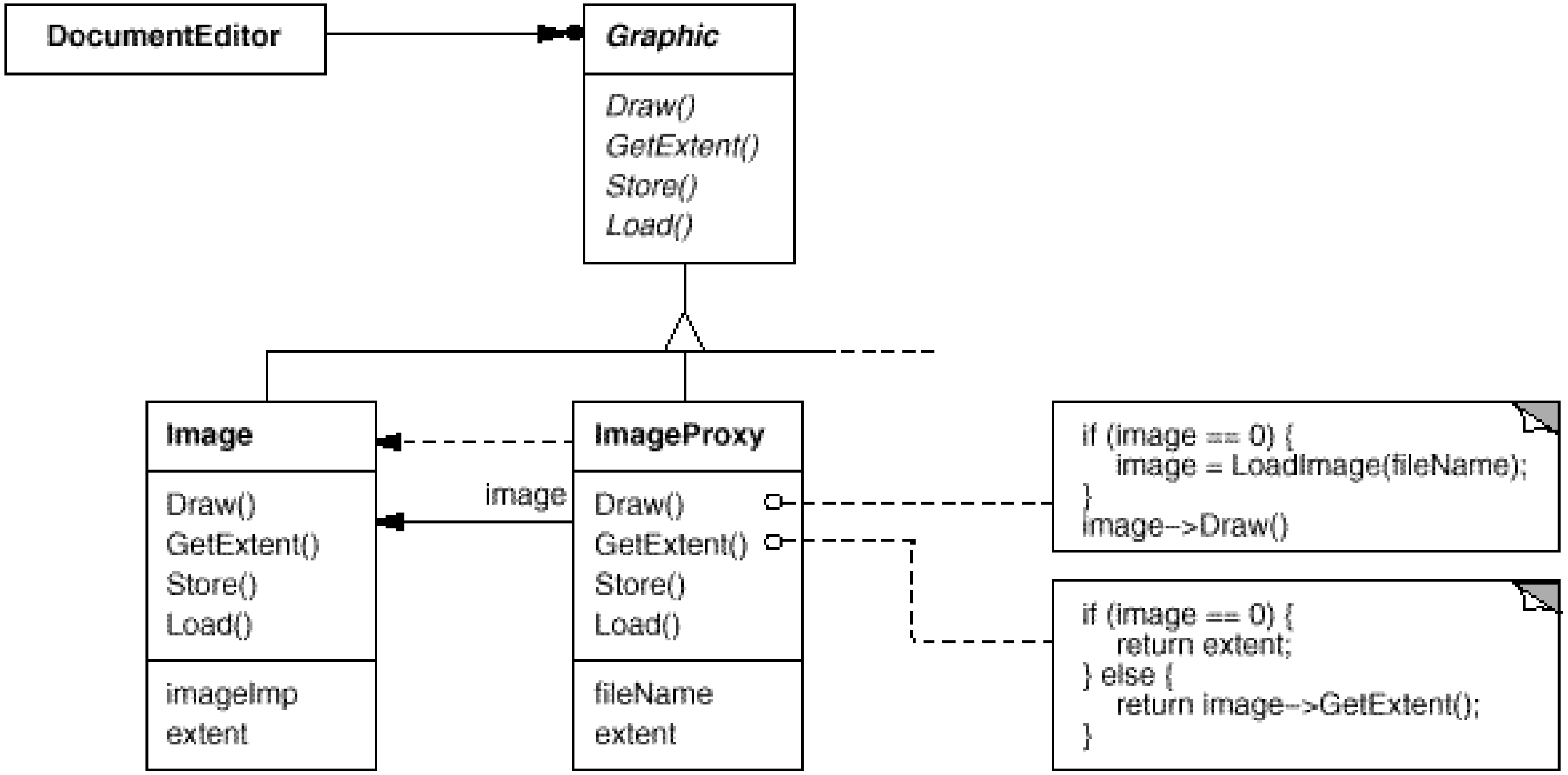
The Proxy Pattern: 3 Types

- **Caching of information ("Remote Proxy")**
  - The Proxy object is a local representative for an object in a different address space
  - Good if information does not change too often
- **Standin ("Virtual Proxy")**
  - Object is too expensive to create or too expensive to download.
  - Good if the real object is not accessed too often
- **Access control ("Protection Proxy")**
  - The proxy object provides protection for the real object
  - Good when different actors should have different access and viewing rights for the same object
  - Example: Grade information accessed by administrators, teachers and students.

Proxy Pattern Class Diagram



Example



```
if (image == 0) {
    image = LoadImage(fileName);
}
image->Draw()
```

```
if (image == 0) {
    return extent;
} else {
    return image->GetExtent();
}
```

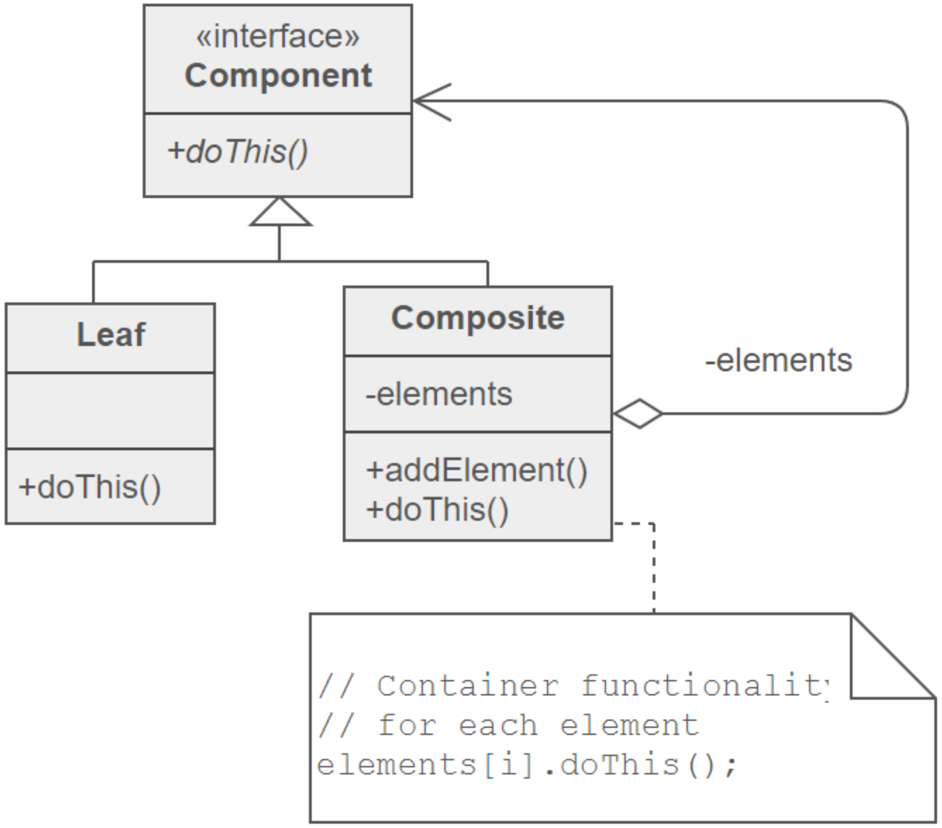
(3) Composite

Composite Pattern

- **Problem:**
  - Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects.
  - Processing of a primitive object is handled one way, and processing of a composite object is handled differently.
  - Having to query the "type" of each object before attempting to process it is not desirable.
- **Intent:** Compose objects into tree structures to represent whole-part hierarchies.
  - Composite lets clients treat individual objects and compositions of objects uniformly.
  - Recursive composition
  - "Directories contain entries, each of which could be a directory."
  - 1-to-many "has a" up the "is a" hierarchy

Composite Pattern

- Menus that contain menu items, each of which could be a menu.
- Row-column GUI layout managers that contain widgets, each of which could be a row-column GUI layout manager.
- Directories that contain files, each of which could be a directory.
- Containers that contain Elements, each of which could be a Container.



```
// Container functionalit
// for each element
elements[i].doThis();
```

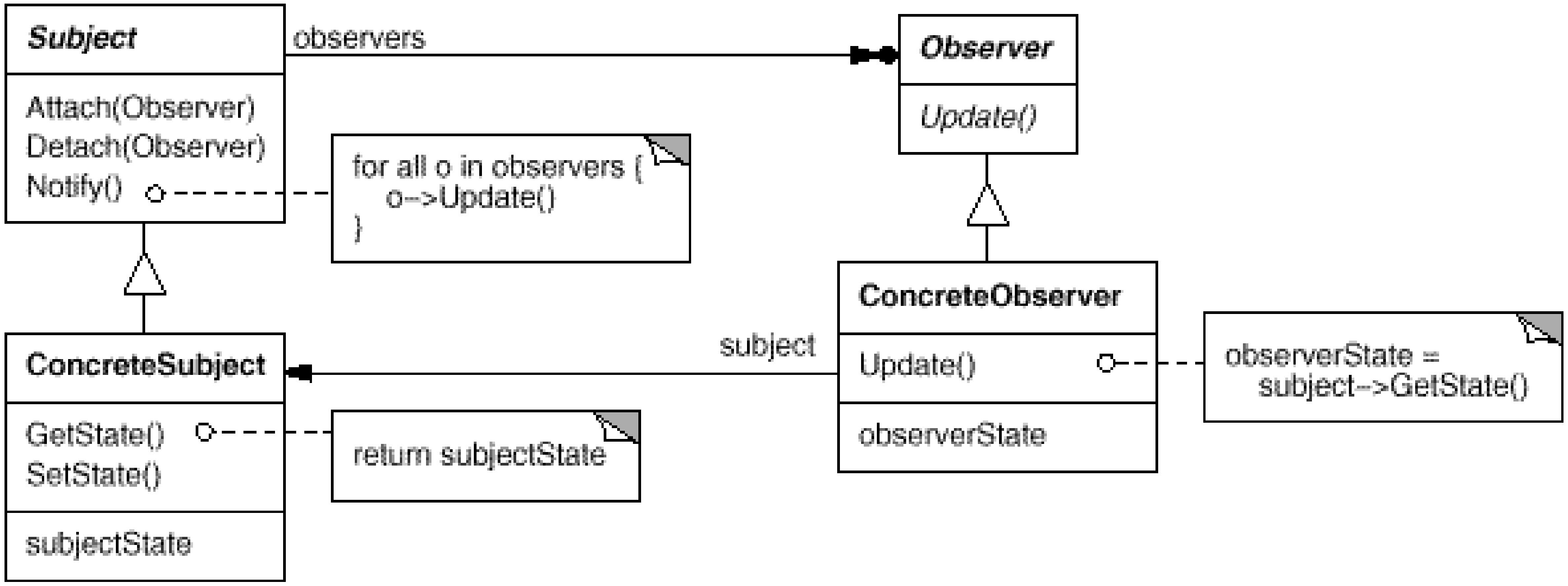
3 Behavioral patterns

(1) Observer

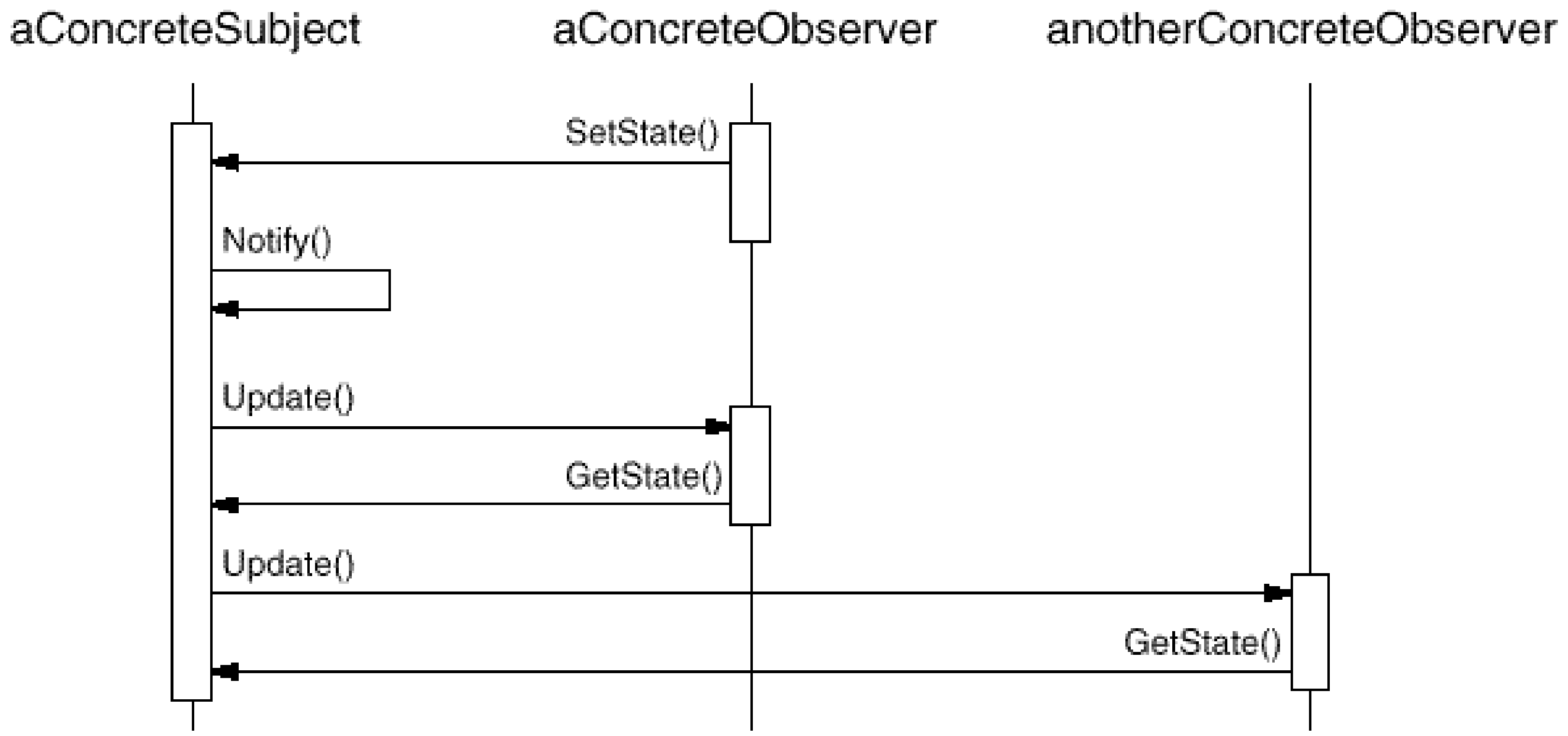
Observer pattern

- **Problem: Dependent's state must be consistent with master's state**
- **Solution: Define four kinds of objects:**
  - Abstract subject: maintain list of dependents; notifies them when master changes
  - Abstract observer: define protocol for updating dependents
  - Concrete subject: manage data for dependents; notifies them when master changes
  - Concrete observers: get new subject state upon receiving update message

Observer pattern



Use of Observer pattern



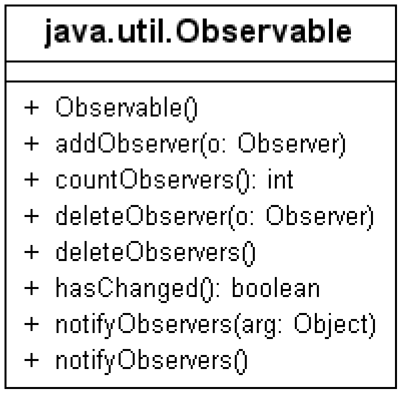
Observer Pattern

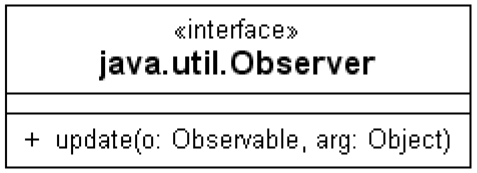
- **Models a 1-to-many dependency between objects**
  - Connects the state of an observed object, the subject with many observing objects, the observers
- **Usage:**
  - Maintaining consistency across redundant states
  - Optimizing a batch of changes to maintain consistency
- **Three variants for maintaining the consistency:**
  - Push Notification: Every time the state of the subject changes, all the observers are notified of the change
  - Push-Update Notification: The subject also sends the state that has been changed to the observers
  - Pull Notification: An observer inquires about the state the of the subject
- **Also called Publish-subscribe.**

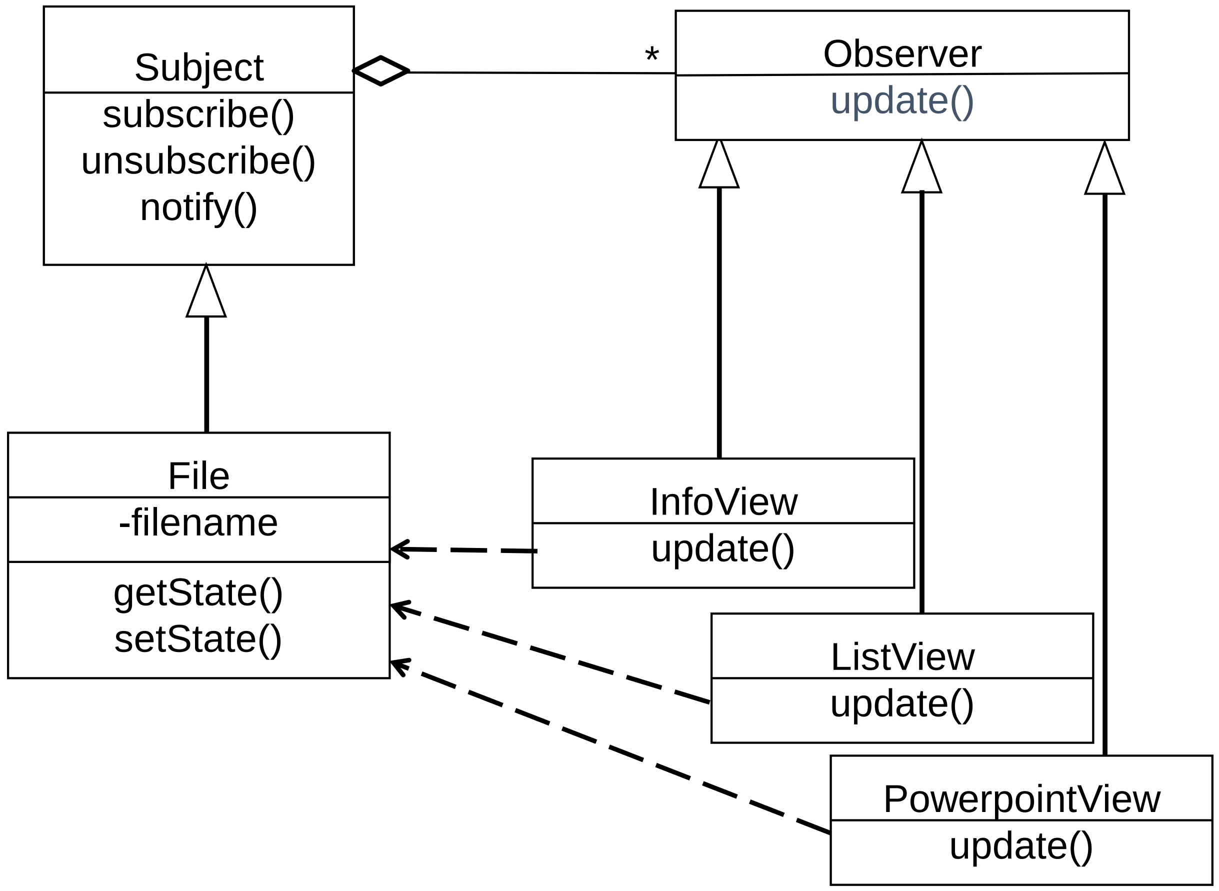
Observer Pattern

- **Advantage:**
  - Low coupling between subject and observers: Subject unaware of dependents
  - Support for broadcasting: Dynamic addition and removal of observers
  - Unexpected updates: No control by the subject on computations by observers
- **Implementation issues**
  - Storing list of observers: typically in subject
  - Observing multiple subjects: typically add parameters to update()
  - Who triggers update: state-setting operations of subject

Example: to Maintain Consistency across Views



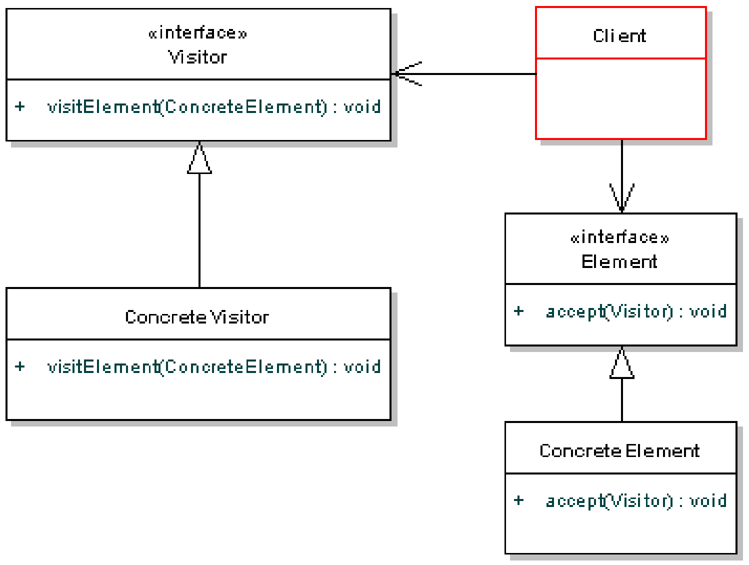




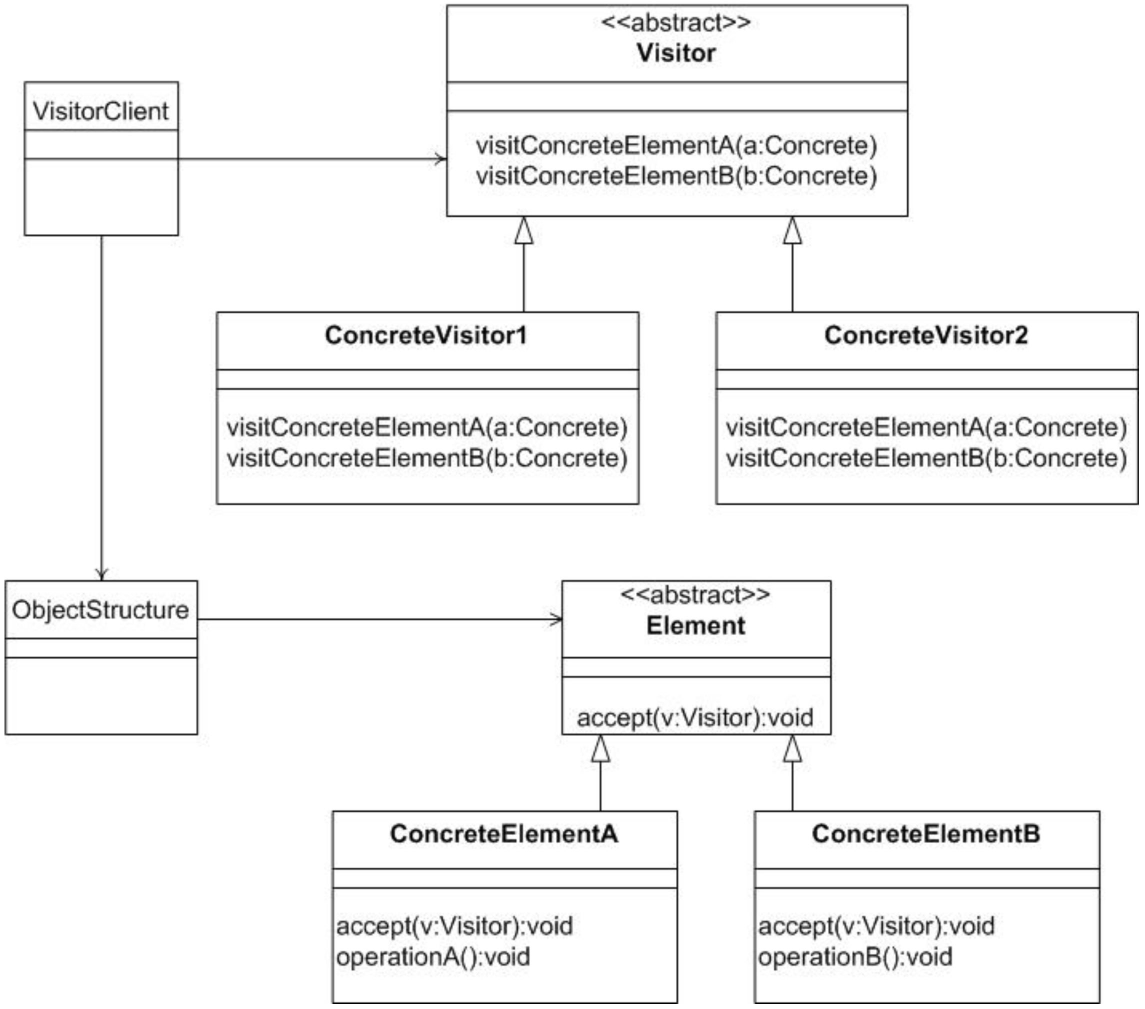
(2) Visitor

Visitor Pattern

- **Visitor pattern: Allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure.**
  - What the Visitor pattern actually does is to create an external class that uses data in the other classes.
  - If the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes.



Visitor Pattern



Example

```
/* Abstract element interface (visitable) */
public interface ItemElement {
   public void accept(ShoppingCartVisitor visitor);
}

/* Concrete element */
public class Book implements ItemElement{
  private double price;
  public void accept(ShoppingCartVisitor visitor){
      visitor.visit(this);
  }
  …
}

public class Fruit implements ItemElement{
  private double weight;
  public void accept(ShoppingCartVisitor visitor){
      visitor.visit(this);
  …
}
```

Example

```
/* Abstract visitor interface */
public interface ShoppingCartVisitor {
   int visit(Book book);
   int visit(Fruit fruit);
}
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {
   public int visit(Book book) {
     int cost=0;
     if(book.getPrice() > 50){
        cost = book.getPrice()-5;
     }else
        cost = book.getPrice();
     System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost ="+cost);
     return cost;
   }
   public int visit(Fruit fruit) {
     int cost = fruit.getPricePerKg()*fruit.getWeight();
     System.out.println(fruit.getName() + " cost = "+cost);
     return cost;
   }
}
```

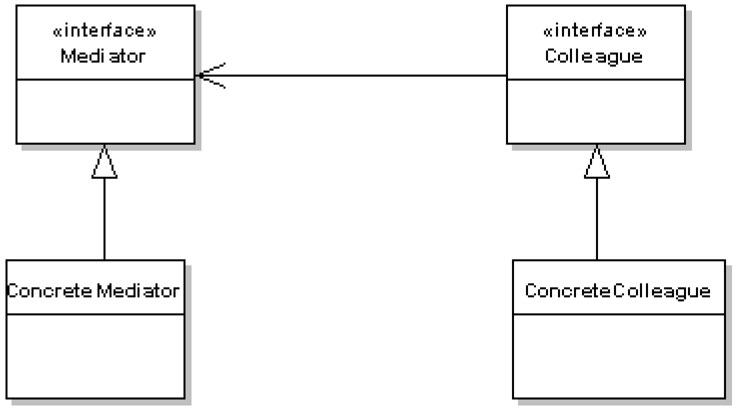
Example

```
public class ShoppingCartClient {

   public static void main(String[] args) {

      ItemElement[] items = new ItemElement[]{
                new Book(20, "1234"),new Book(100, "5678"),
                new Fruit(10, 2, "Banana"), new Fruit(5, 5, "Apple")};

        int total = calculatePrice(items);
        System.out.println("Total Cost = "+total);
      }

      private static int calculatePrice(ItemElement[] items) {
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item : items)
           sum = sum + item.accept(visitor);
        return sum;
      }
}
```

(3) Mediator

Mediator Pattern

- An airport control tower looks after the flights that can take off and land - all communications are done from the airplane to control tower, rather than having plane-to-plane communication.
- This idea of a central controller is one of the key aspects to the **mediator pattern**.
- **Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other.**
- **Allows for the actions of each object set to vary independently of one another.**



Mediator Pattern

- The **Mediator** defines the interface for communication between **Colleague** objects**.**
- The **ConcreteMediator** implements the Mediator interface and coordinates communication between **Colleague** objects.
  - It is aware of all the Colleagues and their purpose with regards to inter communication.
- The **ConcreteColleague**communicates with other colleagues through the mediator.

- Without this pattern, all of the Colleagues would know about each other, leading to high coupling.
- By having all colleagues communicate through one central point we have a decoupled system while maintaining control on the object's interactions.

Example

```
//Mediator interface
public interface Mediator {
  public void send(String message, Colleague colleague);
}

//Colleage interface
public abstract Colleague{
  private Mediator mediator;
  public Colleague(Mediator m) {
    mediator = m;
  }
  //send a message via the mediator
  public void send(String message) {
    mediator.send(message, this);
  }
  //get access to the mediator
  public Mediator getMediator() {return mediator;}
  public abstract void receive(String message);
}
```

Example

```
// Concrete mediator implementation
public class ApplicationMediator implements Mediator {
  private ArrayList<Colleague> colleagues;
  public ApplicationMediator() {
    colleagues = new ArrayList<Colleague>();
  }
  public void addColleague(Colleague colleague) {
    colleagues.add(colleague);
  }
  public void send(String message, Colleague originator) {
    //let all other screens know that this screen has changed
    for(Colleague colleague: colleagues) {
      //don't tell ourselves
      if(colleague != originator) {
        colleage.receive(message);
      }
    }
  }
}
```

```
//Concrete colleage implementation
public class ConcreteColleague extends Colleague {
  public void receive(String message) {
    System.out.println("Colleague Received: " + message);
  }
}
```

Example

```
public class Client {

  public static void main(String[] args) {

    ApplicationMediator mediator = new ApplicationMediator();
    ConcreteColleague desktop = new ConcreteColleague(mediator);
    Colleague mobile = new MobileColleague(mediator);

    mediator.addColleague(desktop);
    mediator.addColleague(mobile);

    desktop.send("Hello World");
    mobile.send("Hello");
  }
}
```

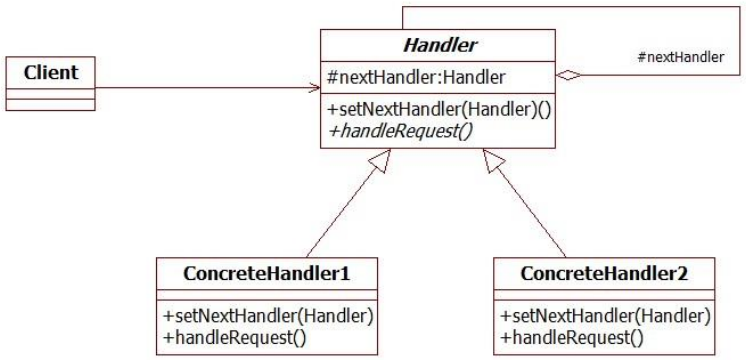
(4) Chain of Responsibility

Chain of Responsibility

- **The chain of responsibility pattern** allows you to pass a request to from an object to the next until the request is fulfilled.
- **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.
  - E.g., you can pass a mortgage application request to a bank manager, and if the manager cannot approve the loan, it can be passed to his supervisor and so on.
- **Applicability:**
  - More than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
  - You want to issue a request to one of several objects without specifying the receiver explicitly.
  - The set of objects that can handle a request should be specified dynamically.

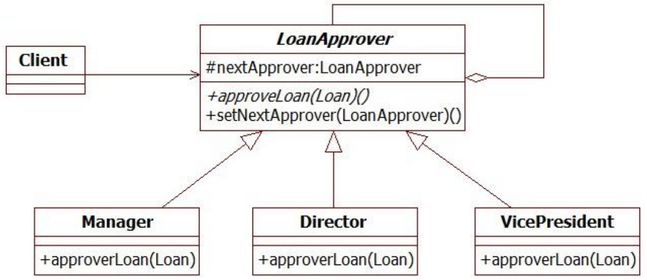
Chain of Responsibility

- The *Handler class* is the parent abstract class for all the objects that can handle requests:
  - The *nextHandler* variable is a reference that points to the next handler. If the request cannot be processed by the object it will be passed to the *nextHandler* for processing.
- The *ConcreteHandler* are the concrete child classes that handles the requests.
  - In its *handelRequest* method it checks to see if it can process the request, if yes it will process the request and return, if not the request will be passed to the next handler. This logic is repeated until the request is fulfilled.



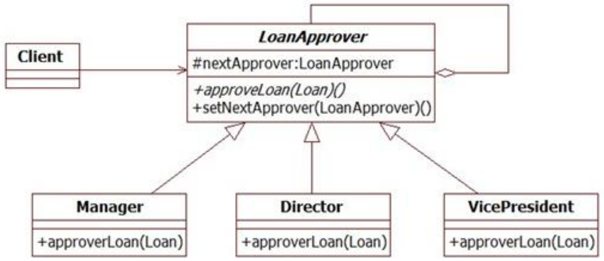
Example

- Let's apply the pattern to an example. In a bank where the approval route for mortgage applications are from the bank manager to the director then to the vice president, where the approval limits are:
  - Manager – 0 to 100k
  - Director – 100k to 250k
  - Vice President – anything above 250k



```
public abstract class LoanApprover{
    protected LoanApprover nextApprover;
    public void setNextApprover(LoanApprover nextApprover){
         this.nextApprover = nextApprover;
    }
    public abstract void approveLoan(Loan i);
}
```

Example

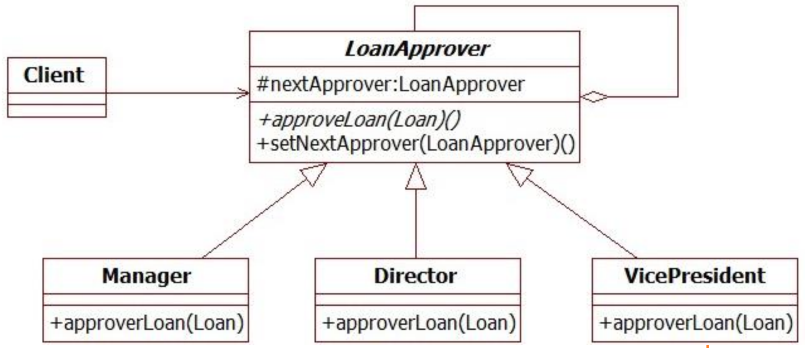


```
public class Director extends LoanApprover {
    public void approveLoan(Loan i) {
        if (i.getAmount() <= 250000)
            System.out.println("Loan amount of " + i.getAmount() +
                                   " approved by the Director");
        else
            nextApprover.approveLoan(Loan i);
        }
}
```

```
public class Manager extends LoanApprover {
    public void approveLoan(Loan i){
        //similar code to Director
    }
}
```

```
public class VicePresident extends LoanApprover {
    public void approveLoan(Loan i){
          System.out.println("Loan amount of " + i.getAmount() + "
                                  approved by the Vice President");
    }
}
```

Example



```
public static void main(String[] args) {
    LoanApprover a = new Manager();
    LoanApprover b = new Director();
    LoanApprover c = new VicePresident();
    a.setNextApprover(b);
    b.setNextApprover(c);
    a.approveLoan(new Loan(50000));  //approved by the manager
    a.approveLoan(new Loan(200000));  //approved by the director
    a.approveLoan(new Loan(500000));  //approved by the vice president
}

The result:
Loan amount of 50000 approved by the Manager
Loan amount of 200000 approved by the Director
Loan amount of 500000 approved by the Vice President
```

(5) Command

Command Pattern

- **Intent**
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Motivation**
  - Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.
  - For example, user interface toolkits include objects like buttons and menus that carry out a request in response to user input. But the toolkit can't implement the request explicitly in the button or menu, because only applications that use the toolkit know what should be done on which object. As toolkit designers we have no way of knowing the receiver of the request or the operations that will carry it out.
  - The Command pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object. This object can be stored and passed around like other objects.

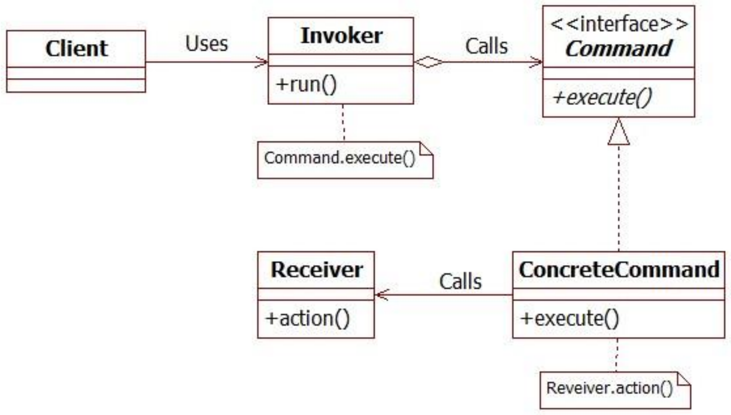
Command Pattern

- **Applicability**
  - Parameterize objects by an action to perform. E.g., callback function.
  - Specify, queue, and execute requests at different times.
  - Support undo. The Command's Execute operation can store state for reversing its effects in the command itself
  - Support logging changes so that they can be reapplied in case of a system crash.
  - Structure a system around high-level operations built on primitives operations.

- The benefit of the command pattern is that it hides the details of the actions that needs to be performed, so that the client code does not need to be concerned about the details when it needs to execute the actions. The client code just need to tell the application to execute the command that was stored.

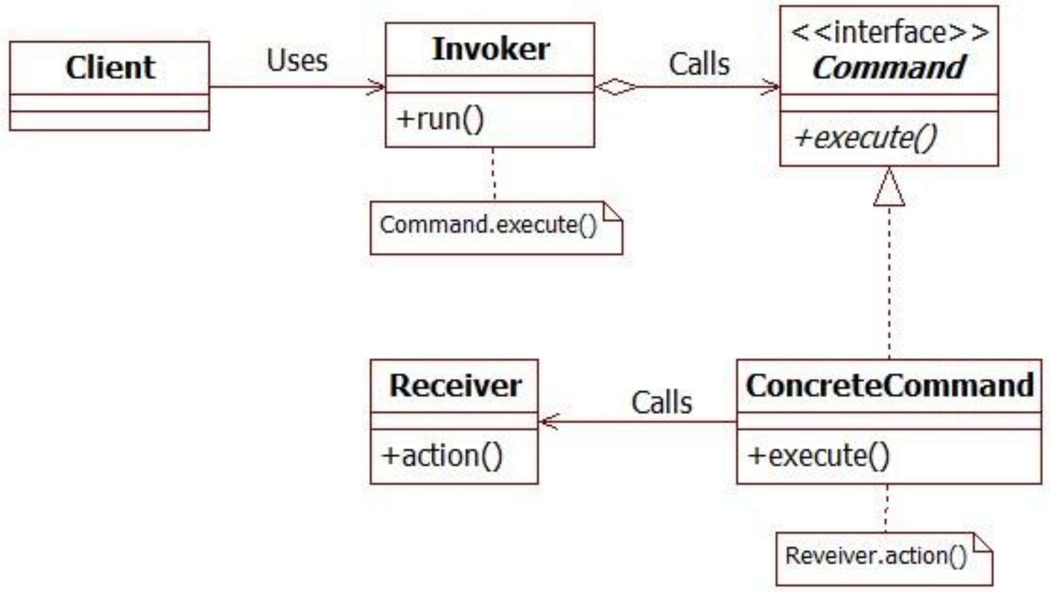
Command Pattern

- The *Command* interface defines the methods that all *ConcreteCommand* classes must implement.
- The *ConcreteCommand* class stores the details of the actions that need to be performed.
- The *Receiver* class performs the action when called upon.
- The *Invoker* class stores the list of commands and can ask the *Command* to execute.
- The *Client* class uses the *Invoker* to run the commands.



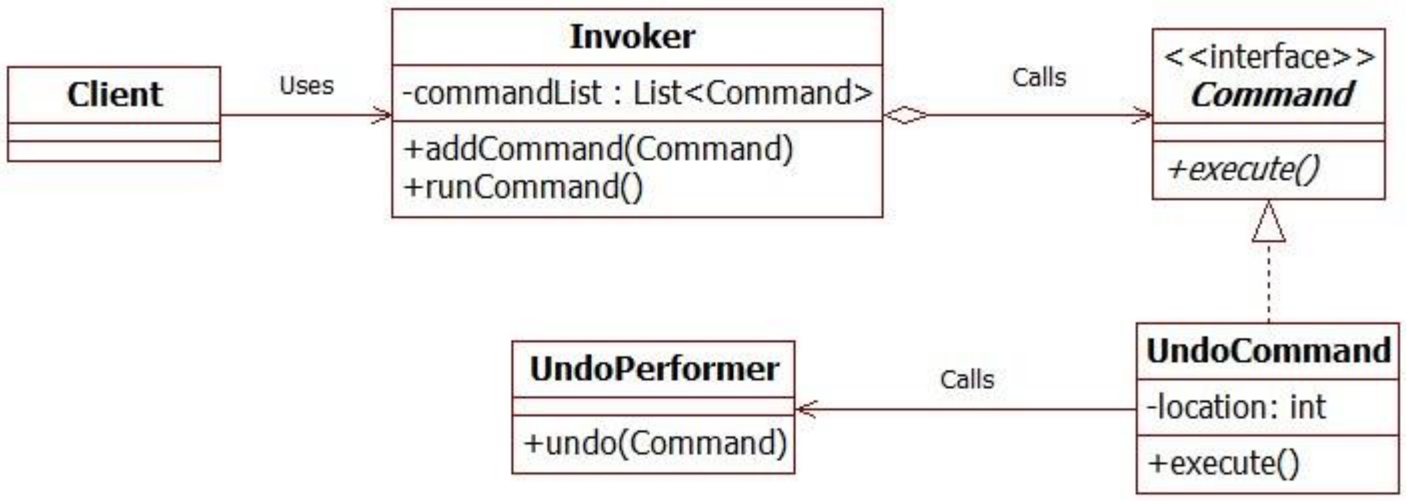
Command Pattern

- The client code (calling code) will used the Invoker to run the commands, where the *Command* objects will call the *Receiver* to perform the action. The benefit is that the client code does not need to know what is stored in the *Command* objects nor the actions that will be performed by the *Receiver*, and this is the key of the Command Design Pattern.

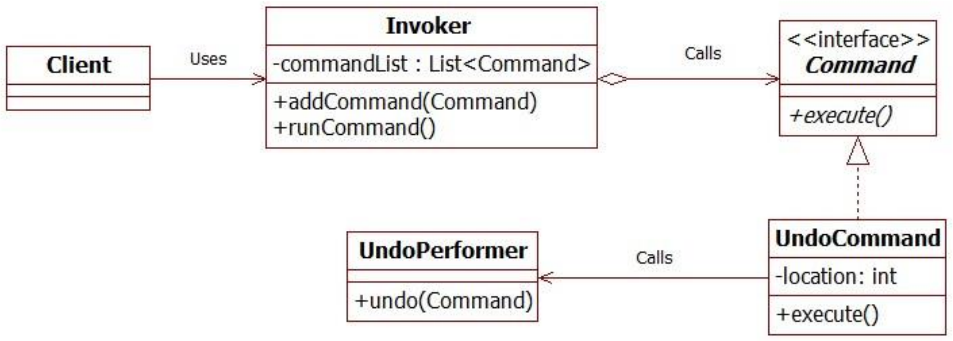


Example

- In our example we need to store some undo actions when the user is using the application, and when the user decides to perform the undo we can just use the invoker to run the commands.

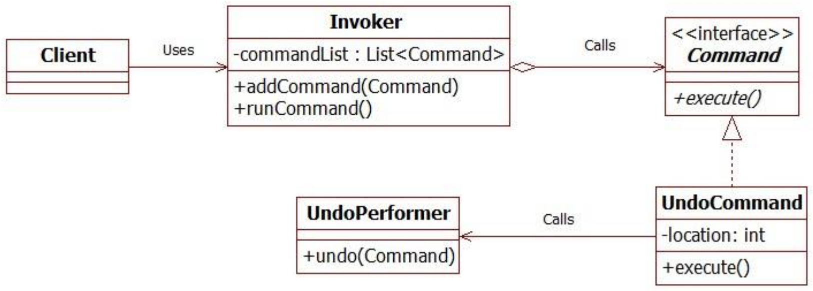


Example



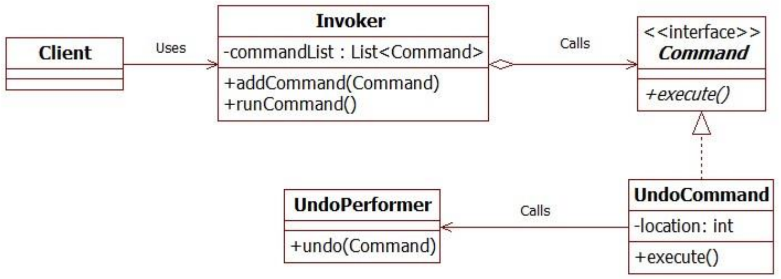
```
public interface Command {
    void execute();
}

public class UndoCommand implements Command {

    private int location;

    public UndoCommand(int originalLocation) {
        location = originalLocation;
    }

    public int getLocation() {
        return location;
    }

    public void execute(){
        new UndoPerformer().undo(this);
    }
}
```

Example



```
public class UndoPerformer {
    public void undo(Command c) {
        if (c.getClass() == UndoCommand.class) {
            int originalLocation = ((UndoCommand) c).getLocation();
            System.out.println("Moving back to position: " +
                                   originalLocation);
        }
    }
}

public class Invoker {
    private Stack<Command> commandList = new Stack<Command>();
    public void runCommand() {
        while (!commandList.isEmpty())
            commandList.pop().execute();
    }

    public void addCommand(Command c) {
        commandList.push(c);
    }
}
```

Example



```
public class Client {
    public static void main(String[] args) {
        Invoker i = new Invoker();

        // save undo to position 100
        Command a = new UndoCommand(100);
        i.addCommand(a);

        // save undo to position 200
        Command b = new UndoCommand(200);
        i.addCommand(b);

         // perform the undo
        i.runCommand(); // the client does not need to know about the
                           details of the undo
        }
}
//The result:
Moving back to position: 200
Moving back to position: 100
```

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

High-level Considerations on Design Patterns for Reusability

Summary

- **Composite, Decorator , Adapter, Bridge, Façade, Proxy (Structural Patterns)**
  - Focus: Composing objects to form larger structures
    - Realize new functionality from old functionality,
    - Provide flexibility and extensibility
- **Command, Observer, Strategy, Template (Behavioral Patterns)**
  - Focus: Algorithms and assignment of responsibilities to objects
    - Avoid tight coupling to a particular solution
- **Abstract Factory, Builder (Creational Patterns)**
  - Focus: Creation of complex objects
    - Hide how complex objects are created and put together

Clues for use of Design Patterns (1)

- **Text: "manufacturer independent", "device independent", "must support a family of products"**

  => Abstract Factory Pattern

- **Text: "must interface with an existing object"**

  => Adapter Pattern

- **Text: "must interface to several systems, some of them to be developed in the future", "an early prototype must be demonstrated"**

  =>Bridge Pattern

- **Text: "must interface to existing set of objects"**

  => Façade Pattern

Clues for use of Design Patterns (2)

- **Text: “complex structure”, “must have variable depth and width”**

  => Composite Pattern
- **Text: “must be location transparent”**

  => Proxy Pattern
- **Text: “must be extensible”, “must be scalable”**

  => Observer Pattern
- **Text: “must provide a policy independent from the mechanism”**

  => Strategy Pattern

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

April 8, 2019