hi-tech college + JOHN BRYCE Leading in IT Education a matrix company

Java Design Patterns

Course Agenda

- Part 1- An introduction
- Part 2 - Some O-O aspects
- Part 3 - Selected Design Patterns

hi-tech college + JOHN BRYCE Leading in IT Education a matrix company

Design Pattern

Part 1- An introduction

Agenda

- Design Pattern- Definition
- History
- Pattern Types
- Learning Design Pattern
- Main Design Patterns
- Frameworks & Architectures vs. DPs

Design Pattern- Definition

- “Design patterns are recurring solutions to design problems you see over and over.” (*The Smalltalk Companion)*James W Cooper

- “Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development.” (Pree 1994)

Design Pattern- What is it for

- How to write a new program feature?
- Have all possible avenues considered?
- A more methodologically way to do the job
- Develop the best solution to the problem.

History

- Design patterns began by Erich Gamma (1992)

***Design Patterns—Elements of Reusable Software*, by Gamma, 1995**

- This book had a powerful impact on those seeking to understand how to use design patterns.

Design Pattern- the need

- Repeating patterns
- Solve the need to recreate solutions from scratch.
- Better modularity and class definitions

Design Pattern- So what is it?

- The interaction between objects.
- Object creation patterns.
- Strategies for object inheritance and containment.

Design patterns existence

- From very low-level specific solutions to broadly generalized system issues.
- Patterns are *discovered* rather than written.
  - We learn from our mistakes
  - We document our best solutions
  - “Pattern mining”

Learning Design Patterns

- Regardless of the language used to write the code, learning design patterns is a multiple-step process.
  1. Acceptance
  2. Recognition
  3. Internalization

Abuse of Patterns

- Beware of unnecessary generalization
- Obscuring structure
- Performance overhead
- Effort required to identify and adapt patterns (value/effort)
- Panacea or "silver bullet“
- Restricted to software design or Object-Oriented design

Patterns Types

- In "*Design Patterns"*, on a middle level of generality, are cross application areas and encompass several objects.
- The authors divided these patterns into three types:
  - creational,
  - structural,
  - behavioral.

Creational

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral

- Interpreter
- Template Method
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor

Design Patterns Classification

- First classification: what a pattern does:
  - **Creational patterns** - concern the process of object creation
  - **Structural patterns -** deal with the composition of classes or objects
  - **Behavioral patterns** - characterize the ways classes or objects interact

- Second classification: how the pattern is applied:
  - **Class patterns** - 'is A' relationships, static
  - **Object patterns** - 'has A' relationships
  - **Compound patterns -** deal with recursive object structures

Purpose and scope

		PURPOSE		
		CREATIONAL	STRUCTURAL	BEHAVIORAL
SCOPE	CLASS	Factory Method	Adapter	Interpreter Template Method
	OBJECT	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

purpose: what a pattern does
scope: whether a pattern applies primarily to classes or objects

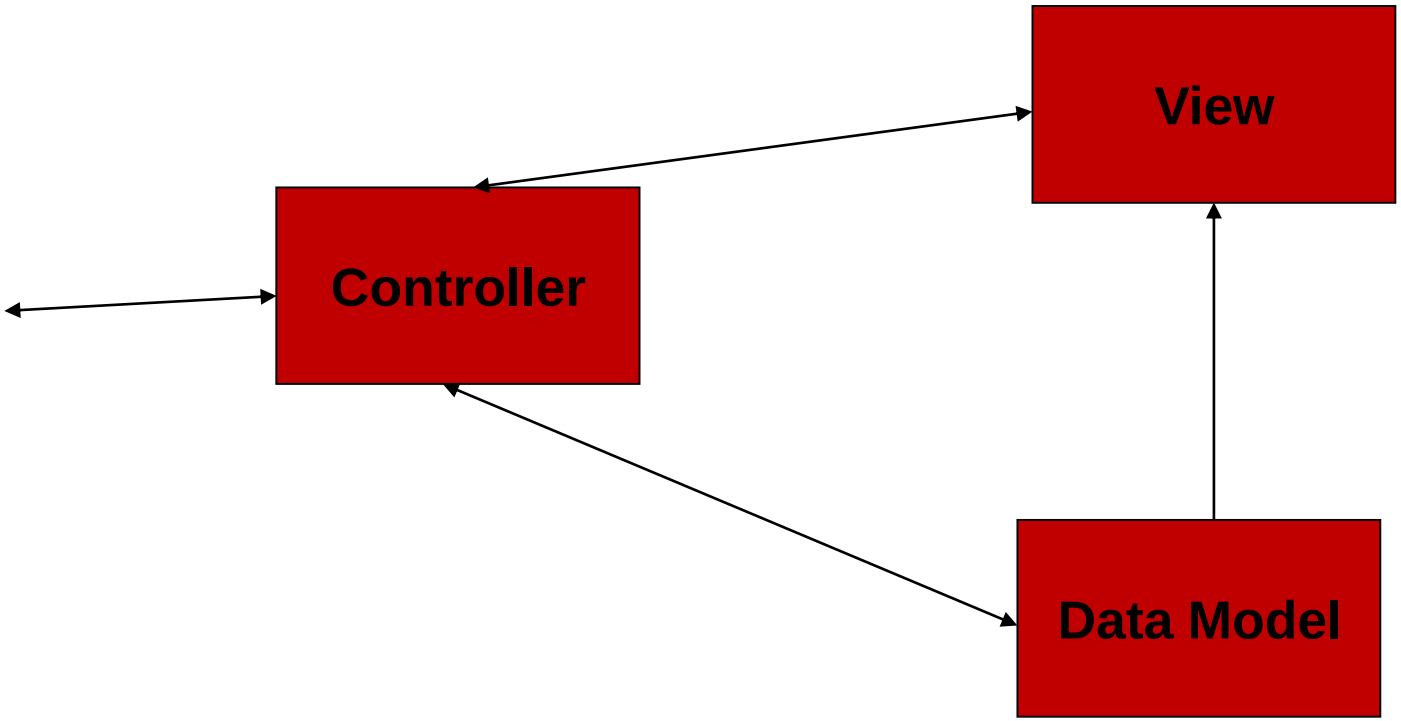
Frameworks vs. DPs

- Wiki:
  - An abstraction in which software providing generic functionality can be selectively changed by additional user written code, thus providing application specific software
- Frameworks reuse design patterns
  - Internal parts or pluggable modules may use design patterns

Architecture vs. DPs

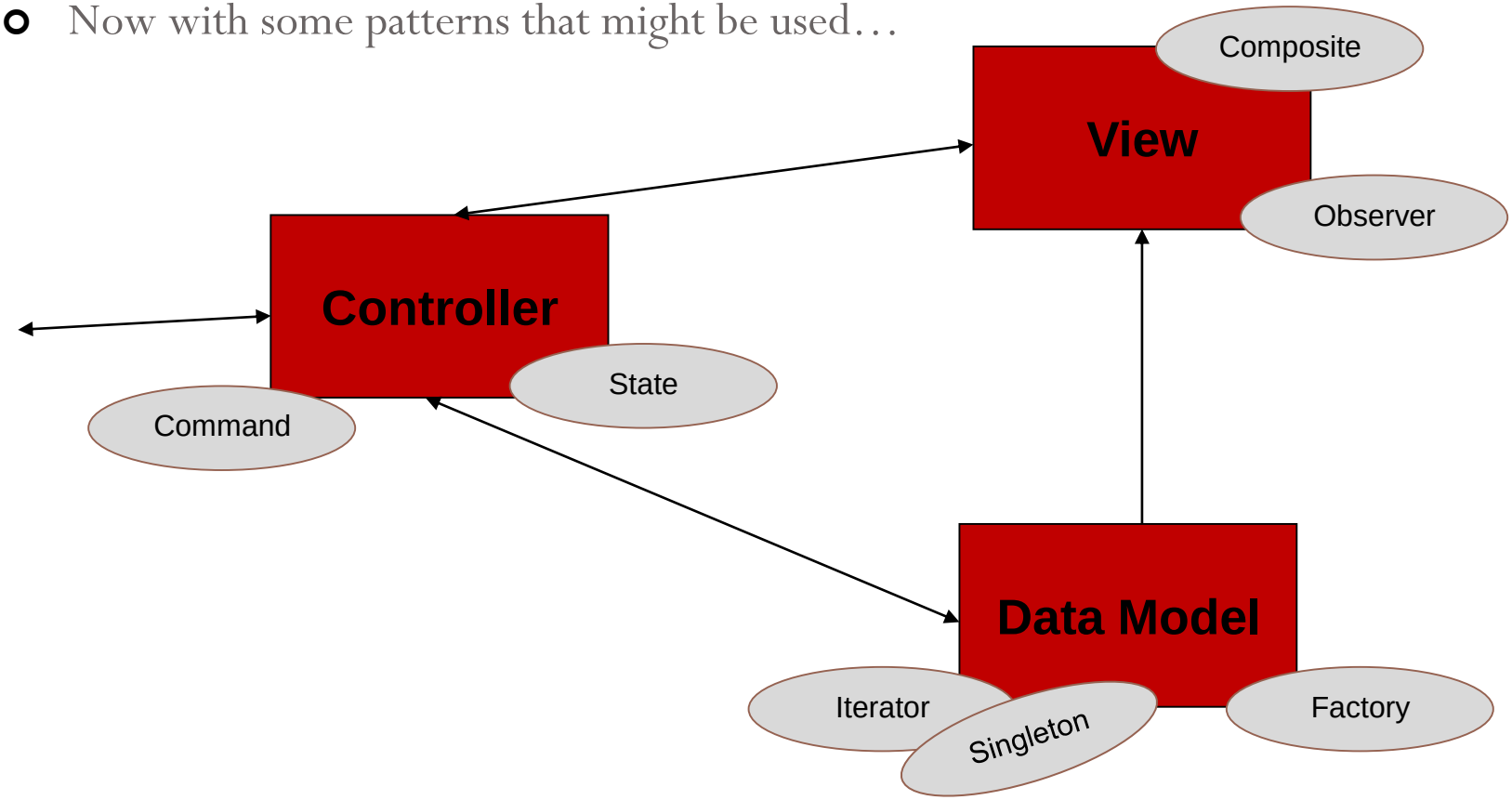
- Software architecture refers to the “bigger picture” of an application
  - It is still very specific – but focuses on flow, use-cases, core components and packages
  - Many design patterns might be used in order to implement the architecture via OOP
- Architecture examples:
  - MVC – Model View Controller
  - 2-Tier, 3-Tier and N-Tier

Model-View-Controller Architecture



Model-View-Controller Architecture

- Now with some patterns that might be used…



hi-tech college + JOHN BRYCE Leading in IT Education a matrix company

Design Pattern part II

Some O-O aspects

Agenda

- Object Oriented approach
- Object composition
- Problems of redundancy
- Defining Object roles
- Inheritance vs. Composition

Object-Oriented Approaches

- Keeping classes separated
- Avoid “reinventing the wheel”
- There are a number of strategies that OO use to achieve separation, among them encapsulation and inheritance.

Object Composition

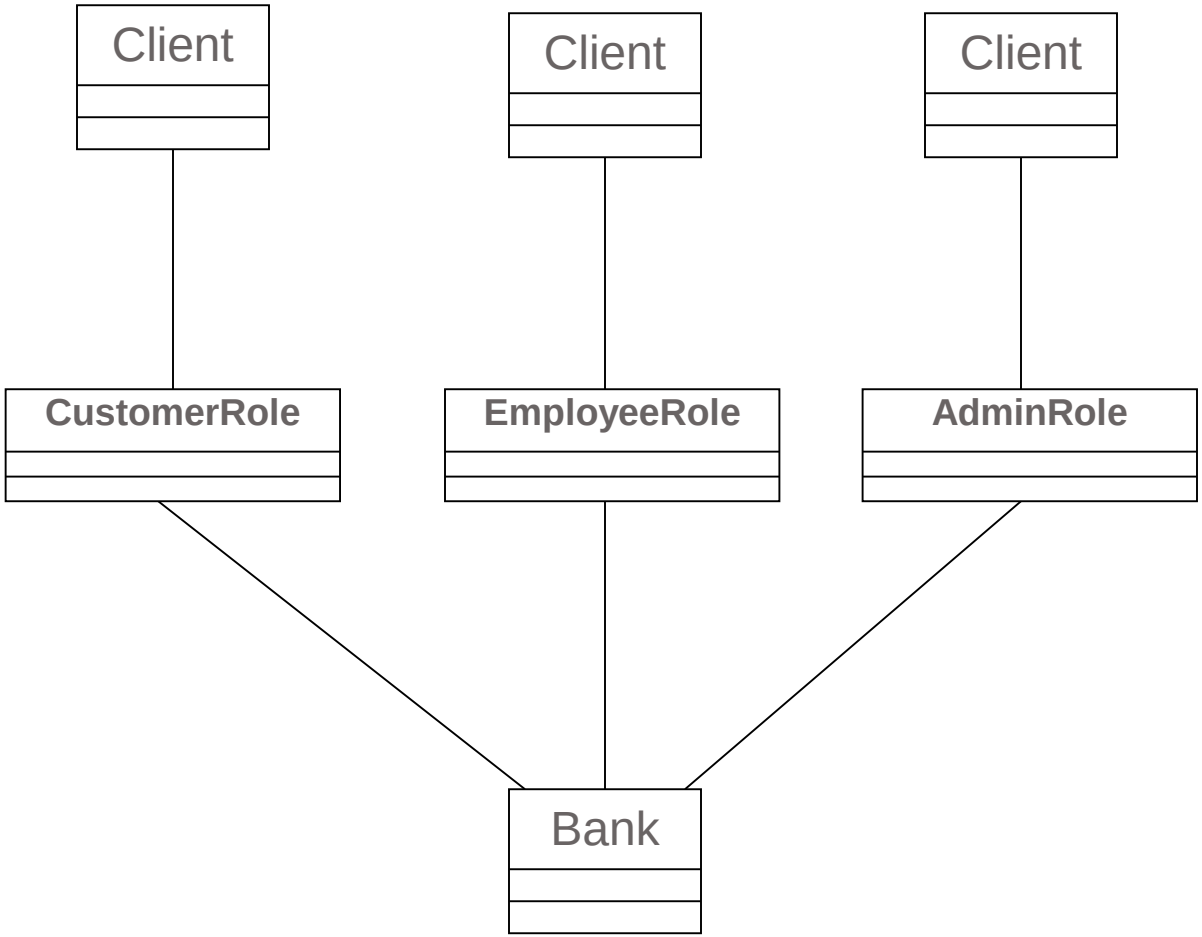
- object composition- “The construction of objects that contain others, encapsulation of several objects inside another one.”
- Inheritance is not the solution of every problem

Problems of Redundancy

- Problem: Many elements in a system will share similar structures and/or functionality
- Common solutions:
  - Subroutines and modules
  - Inheritance
    - Define new structure and/or functionality then define a static relationship with “parents” to use theirs
      - Implementation (operations and attributes)
      - Interface (sub-type)
  - Composition
    - Combine common “parts” then define new structure and/or functionality as necessary

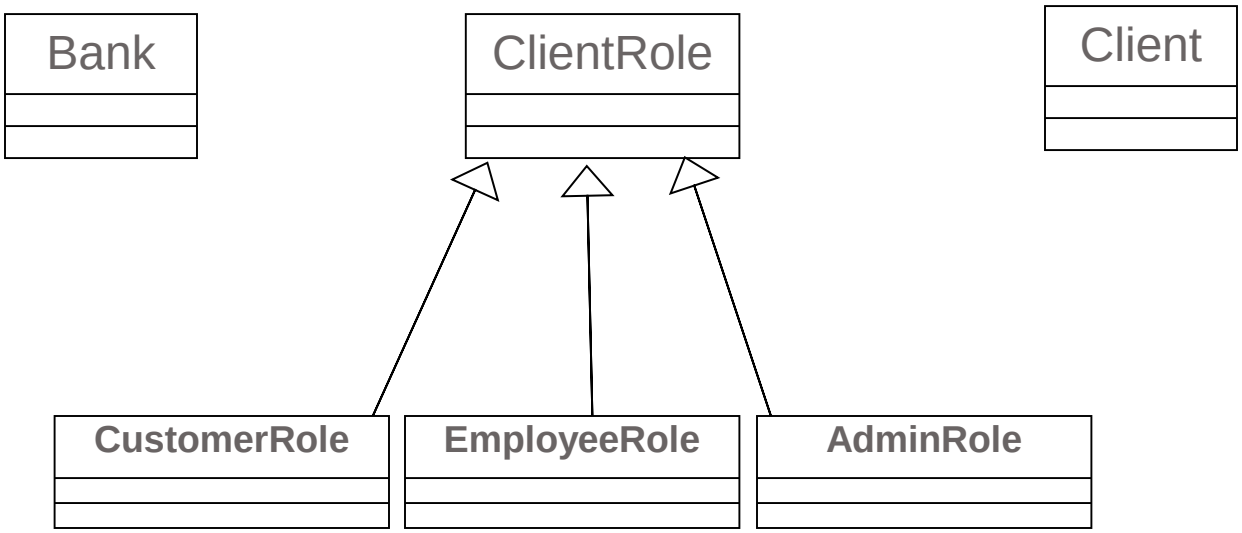
Example - Modules

- Here, every new client type – forces a whole new module creation



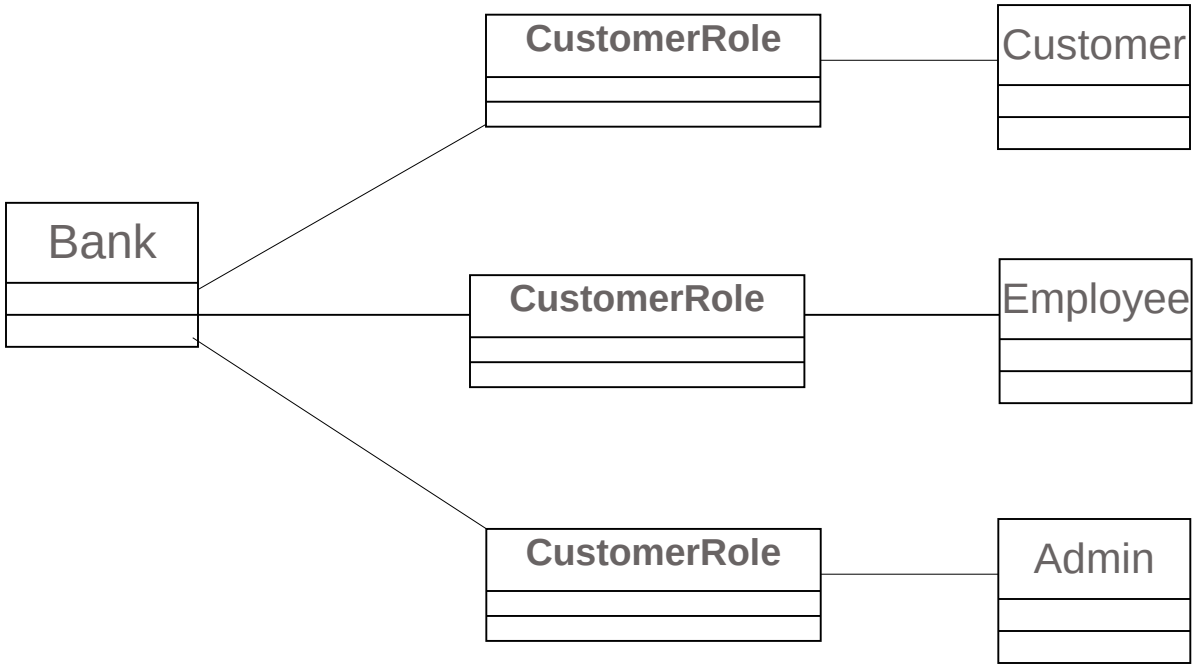
Example - Inheritance

- Here, every new client type is a new subtype of ClientRole



Example - Composition

- Here, client types are visitors of a consistent CustomerRole class used by the Bank



Inheritance vs. Composition

- **White-box reuse** (subclassing): Uses visible implementations
  + easy to use/modify
  - static, tends to breaks encapsulation, limited flexibility

- **Black-box reuse** (composition): Uses hidden implementations
  + dynamic, less dependencies
  - more objects, complex relationships and dependencies

- ***Principle 2:*** Favor composition over inheritance

hi-tech college + JOHN BRYCE Leading in IT Education a matrix company

Design Pattern part III

Selected Patterns

Agenda

- Creational Patterns
  - The Abstract Factory
  - The Factory Method pattern
  - Builder pattern
  - Singleton Pattern
  - Prototype
- Structural Patterns
  - Adapter pattern
  - Facade Pattern
  - The Bridge Pattern
  - Composite Pattern
  - Proxy Pattern
  - Decorator pattern
- Behavioral Patterns
  - Iterator Pattern
  - Strategy Pattern
  - State Pattern
  - Command Pattern
  - Mediator Pattern
  - Observer Pattern
  - Visitor Pattern
  - Chain of responsibility

hi-tech college + JOHN BRYCE Leading in IT Education a matrix company

Creational Patterns

The Abstract Factory

- Abstracts the object creation
- Lets extensibility

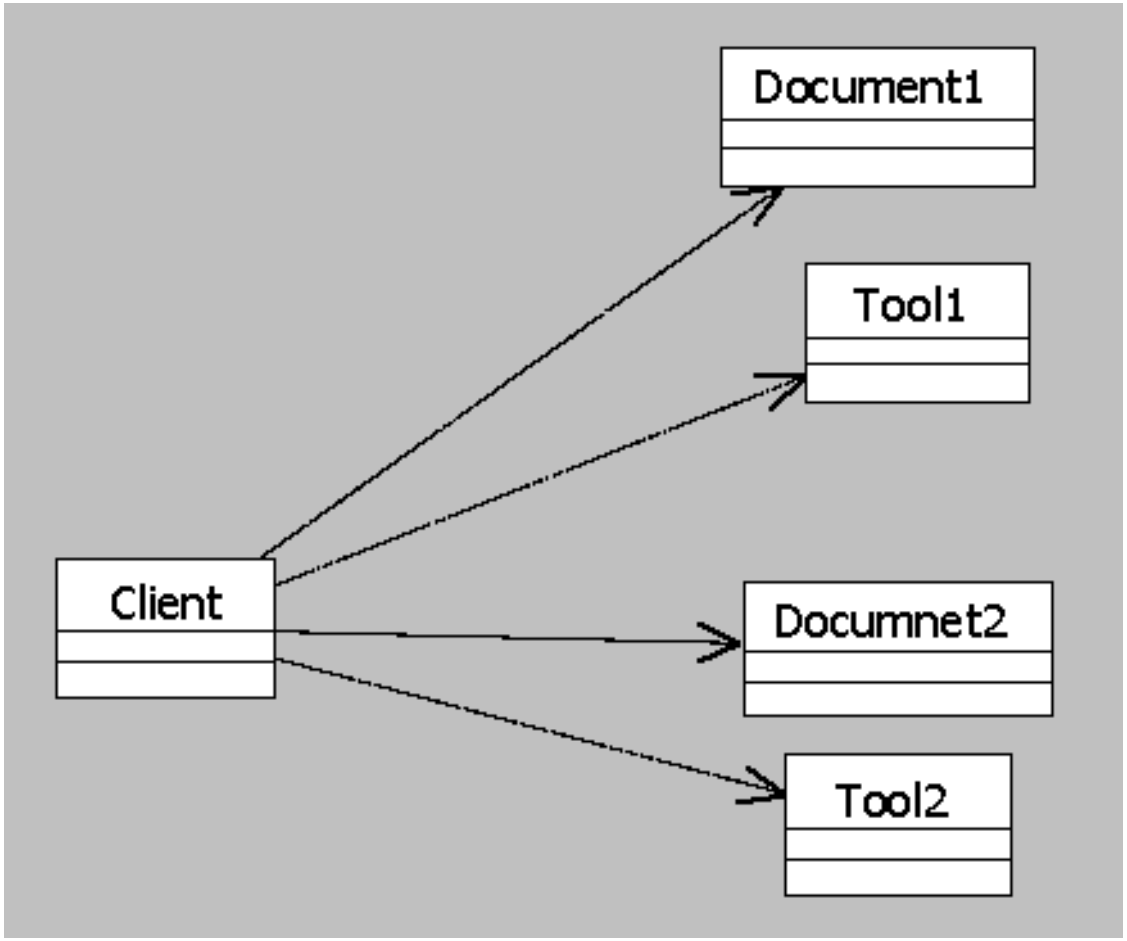
The Abstract Factory -Goal

- Provides an interface for creating families of related or dependent objects.

The Problem

- An office uses 2 kinds of reports. Each report has to be processed before entering the database.
  - Each one of the reports has different fields and needs different treatment
  - A third report is in planning
- What's the easiest way to deal with it (i.e. how do we make it as simple as possible for the client, and for expansion)?

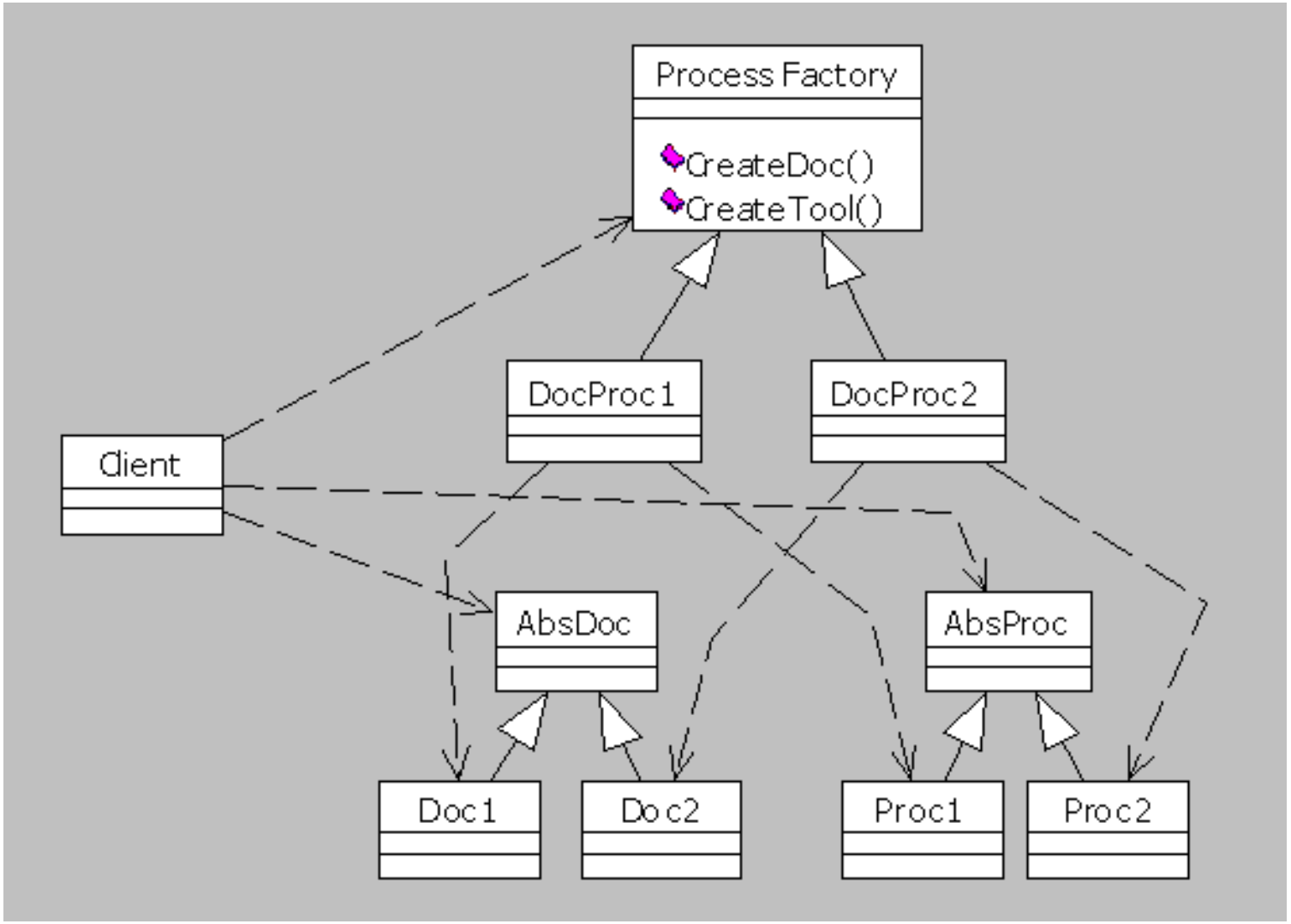
Solution 1



Solution 1 Problems

1. Client must know each existing document and each existing tool types.
2. Client must know the tools methods for dealing with the report.
3. New report enforces us to rewrite the client.

Abstract Factory Solution



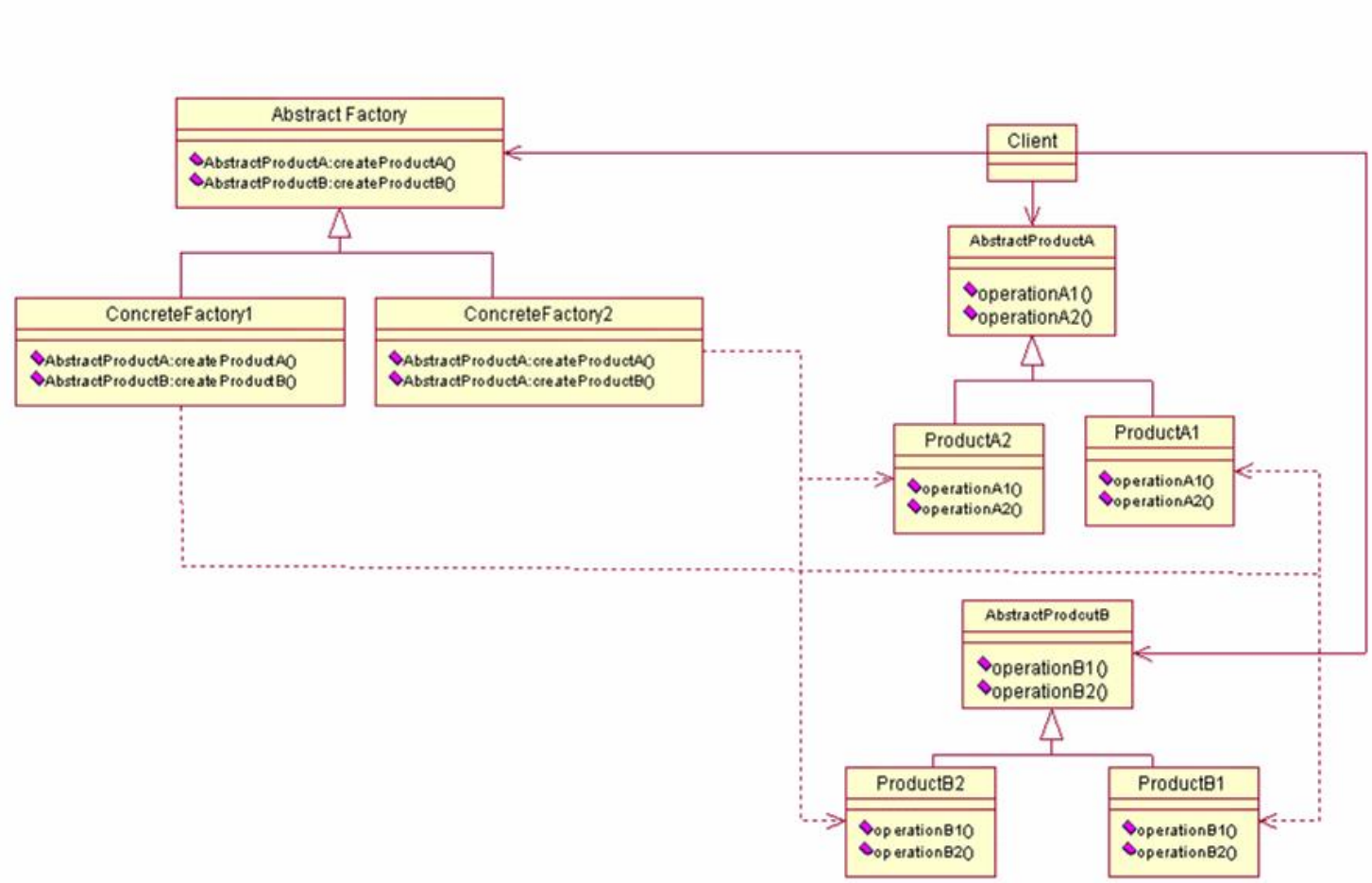
In General Terms

- The related objects are generally derived from the same abstract classes.
- Each family has a similar class hierarchy
- It easy to switch from one hierarchy to another with minimal change in code.

Incentive

- A new kind of products needs only a new concrete factory classes.
- Provides the product creation interface
- Uses ***factory method pattern*** to delegate the real object creation task
- Helps exchanging product families easily
- The concrete product classes are isolated.

Abstract Factory



In Simpler Terms

- For new products family, one new factory method is added in the Concrete Factory.
- One factory method taking the kind of product object
- Generally, the concrete factory is implemented as a singleton

Code Example

```
public interface InterestCalculator {
    public abstract void calculateInterest();
}
```

```
public class InterestAlgorithm1 implements InterestCalculator {
    public void calculateInterest() {
            System.out.println("InterestAlgorithm1.calculateInterest");
    }
}
```

```
public class InterestAlgorithm2 implements InterestCalculator {
    public void calculateInterest() {
            System.out.println("InterestAlgorithm2.calculateInterest");
    }
}
```

Code Example

```
public abstract class Account {
    private InterestCalculator _interestCalculator;

    public abstract void deposit(int amt);

    public Account(InterestCalculator interestCalculator) {
            this._interestCalculator = interestCalculator;
    }
    public void calculateInterest() {
            this._interestCalculator.calculateInterest();
    }
}
```

Code Example

```
public abstract class BankSystemFactory {
    public abstract Account createAccount(InterestCalculator interestCalculator);
    public abstract InterestCalculator createInterestCalculator();
}
```

```
public class BankSystemFactory1 extends BankSystemFactory {
    public Account createAccount(InterestCalculator interestCalculator) {
        return new Savings(interestCalculator);
    }
    public InterestCalculator createInterestCalculator() {
        return new InterestAlgorithm1();
    }
}
```

```
public class BankSystemFactory2 extends BankSystemFactory {
    public Account createAccount(InterestCalculator interestCalculator) {
        return new Checking(interestCalculator);
    }
    public InterestCalculator createInterestCalculator() {
        return new InterestAlgorithm2();
    }
}
```

Code Example

```
public class Checking extends Account {
	public Checking(InterestCalculator interestCalculator) {
		super(interestCalculator);
	}

	public void deposit(int amt) {
		System.out.println("Checking deposit: " + amt);
	}
}
```

Code Example

```
public class Savings extends Account {
	public Savings(InterestCalculator interestCalculator) {
		super(interestCalculator);
	}

	public void deposit(int amt) {
		System.out.println("Savings deposit: " + amt);
	}
}
```

Code Example

```
public class BankSystem {
	public static void main(String args[]) {
		String className = null;
		if (args.length == 0) {
			className = "BankSystemFactory2";
		} else {
			className = args[0];
		}
		try {
			Class cls = Class.forName(className);
			BankSystemFactory factory = (BankSystemFactory) cls.newInstance();
			//Instantiate the BankSystem and run.
			BankSystem bank = new BankSystem(factory);
		} catch (Exception e) {
			System.out.println("The class name " + className + " provided at command line doesn't exist");
		}
	}
	public BankSystem(BankSystemFactory bankFactory) {
		InterestCalculator i = bankFactory.createInterestCalculator();
		Account a = bankFactory.createAccount(i);
		a.deposit(100);
		a.calculateInterest();
	}
}
```

The Factory Method pattern

- Creating objects inside a class with factory method is more flexible than creating an object directly
- It is because there are parallel hierarchies of classes to be created.

Factory Method pattern -Goal

- An interface for creating an object, but let subclasses decide which class
- Factory Method lets a class defer instantiation to sub-classes.

In General Terms

- The object creation is deferred to subclasses when the knowledge of the kind of object to be created is not available.

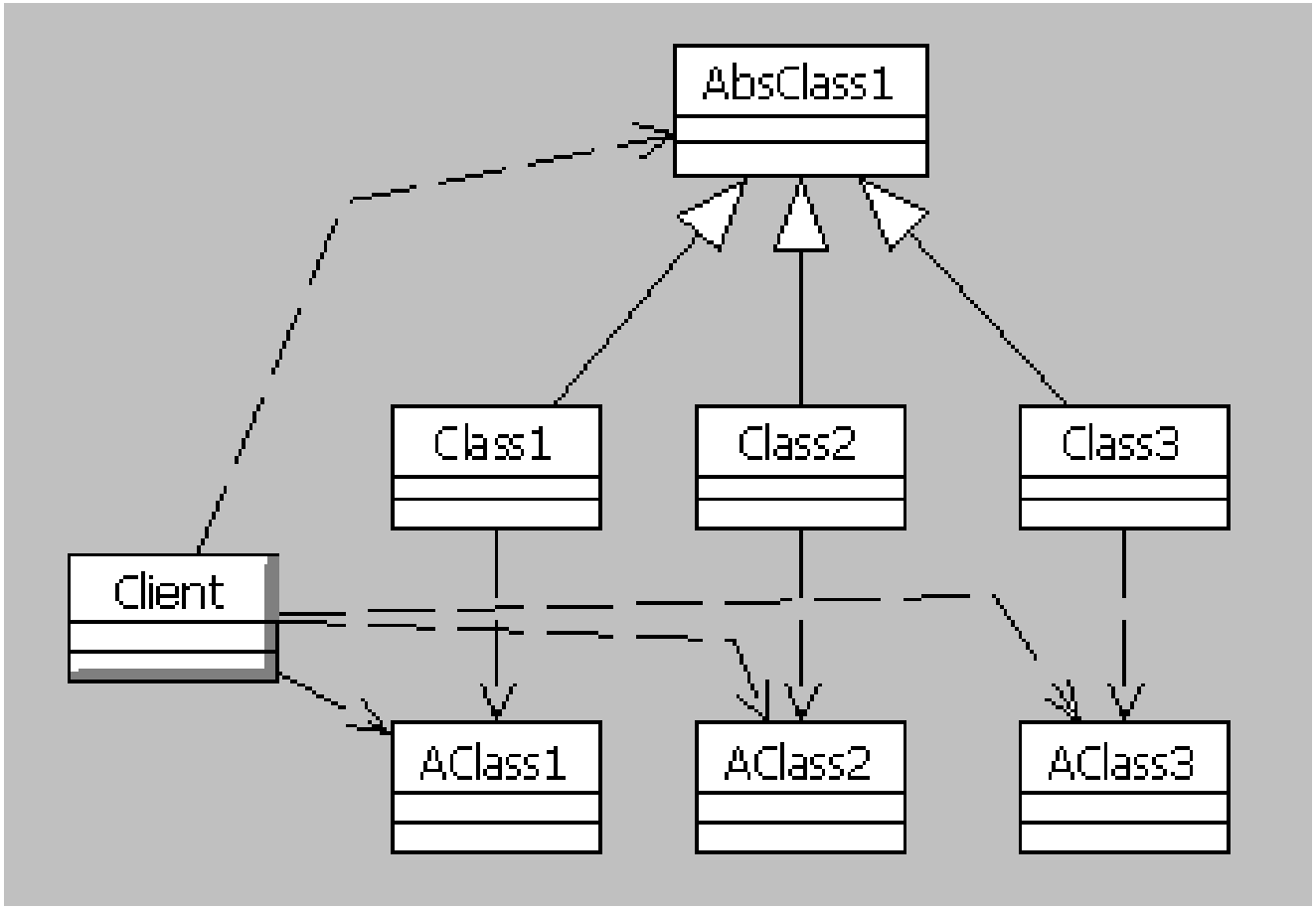
In General Terms

- Frameworks use abstract classes to define and maintain relationship between objects.
- When applications create specific subclasses of the framework abstract classes, they override the factory method and create the specific object required.
- Thus Factory Methods eliminates the need to bind application-specific classes into your framework code.

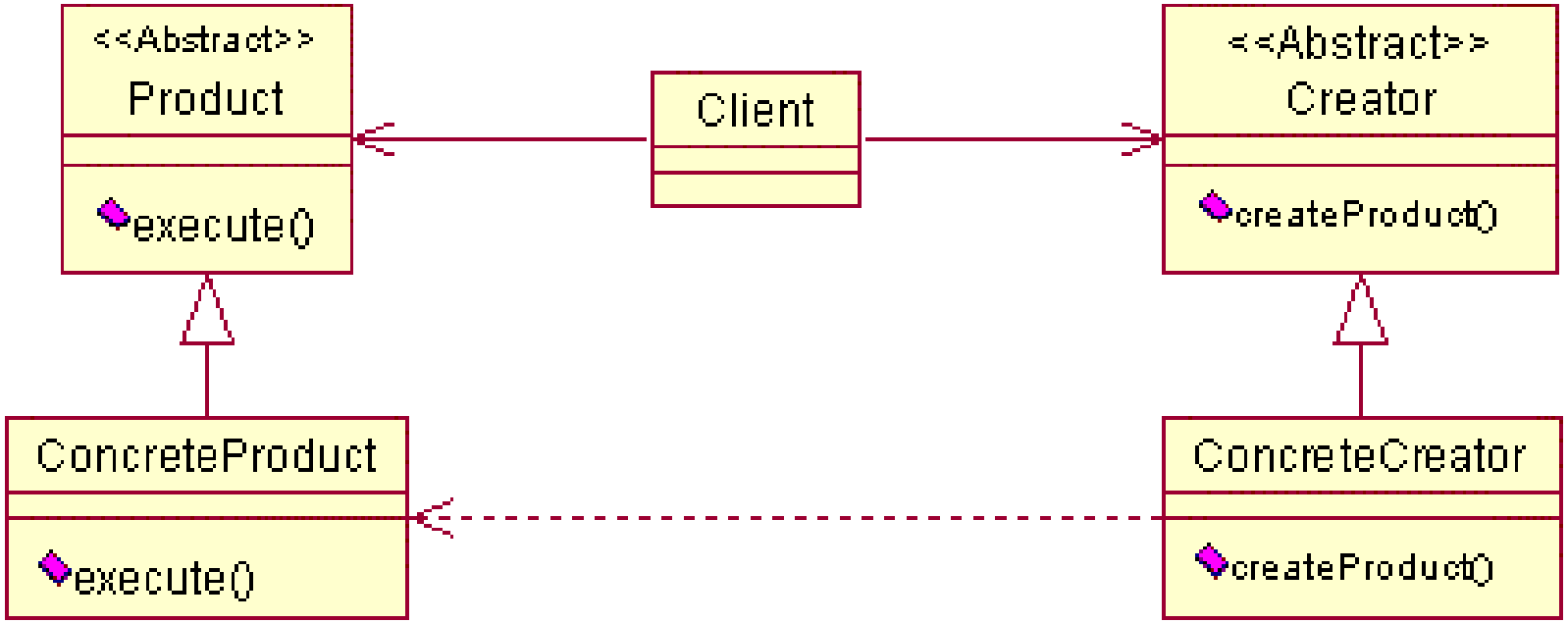
Incentive

- Delegates object creation to a subclass.
- More flexible than creating directly.
- As the Abstract class can provide default object, and the subclass can provide extended version of an object.

The Problem


AbsClass1
Class1
Class2
Class3
Client
AClass1
AClass2
AClass3

The Factory Method Solution



A.K.A

- Also known as Virtual constructor

Related Patterns

- **Abstract Factory**
- **Template**
- **Prototype**

Code Example

```
//Abstract Creator
/* It declares the factory method that returns an object of type Product. */
abstract class Creator{
      //other operations using product
abstract Product createProduct();
}

 //Concrete Creator
/* Overrides the factory method to return an instance of a ConcreteProduct */
class ConcreteCreator extends Creator{
      Product createProduct(){
            return new ConcreteProduct();
      }
}
```

Code Example

```
//Abstract Product
/* This defines the interface of objects that factory
factory method creates
*/
abstract class Product{
	public abstract void execute();
}


/* Implements the Product interface */
class ConcreteProduct extends Product{
	public void execute(){
		System.out.println("Hello Factory Pattern");
	}
}
```

Code Example

```
// Client class
public class Client{
	public static void main(String args[]){
		//May get it through a Factory maker
		ConcreteCreator creator = new ConcreteCreator();
		Product p = creator.createProduct();
		p.execute();
	}
}
```

Java DOM Example

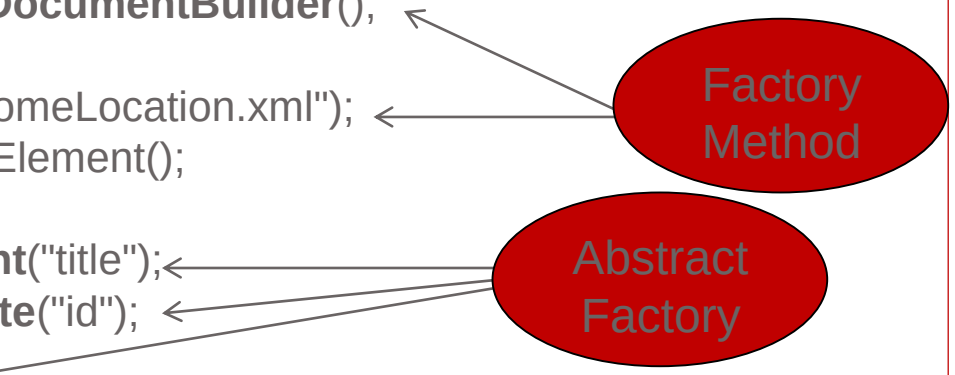
Java DOM uses both Abstract Factory & Factory Method patterns :

```
public static void main(String[] args) throws Exception{
        DocumentBuilderFactory f=DocumentBuilderFactory.newInstance();
        DocumentBuilder parser=f.newDocumentBuilder();

        Document doc=parser.parse(“someLocation.xml");
        Element root=doc.getDocumentElement();

        Element title=doc.createElement("title");
        Attr attribute=doc.createAttribute("id");
        attribute.setValue("8");
        Text text=doc.createTextNode("my new title");

                ….
}
```



Java DOM Example

Java DOM uses both Abstract Factory & Factory Method patterns :

Factory Method:

- DocumentBuilderFactory – creates the Parser
- DocumentBuilder – is the parser that creates DOM Document

Abstract Factory:

- Document creates various of XML entities like
  - Element
  - Attribute
  - Text

The Builder pattern

- We need to use some ‘factory’ to produce objects
- Different types of objects are needed to complete some complex task
- Client need to easily select an object, no matter how complex it is to built

Builder pattern -Goal

- Abstract the steps needed to construct an object
- Different implementations will use these steps to create different representation of objects

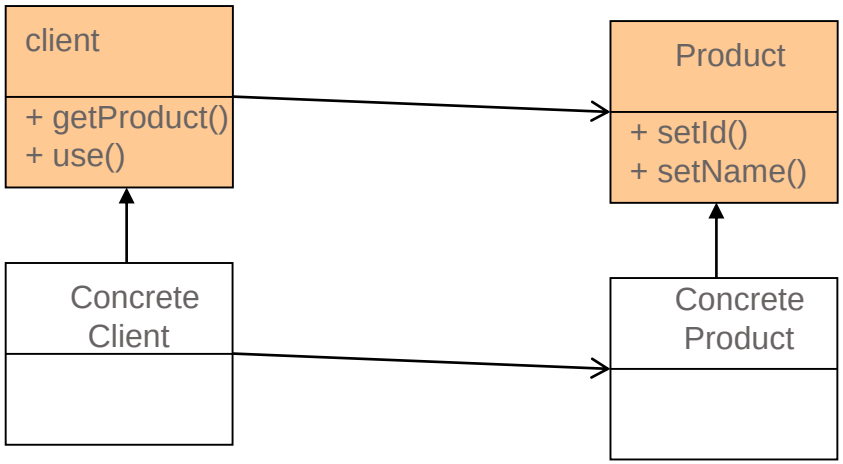
In General Terms

- The object creation is done differently by various implementations
- Each implementation populates a object with different data / configuration
- Director chooses one of these implementations and orchestrates the creation steps

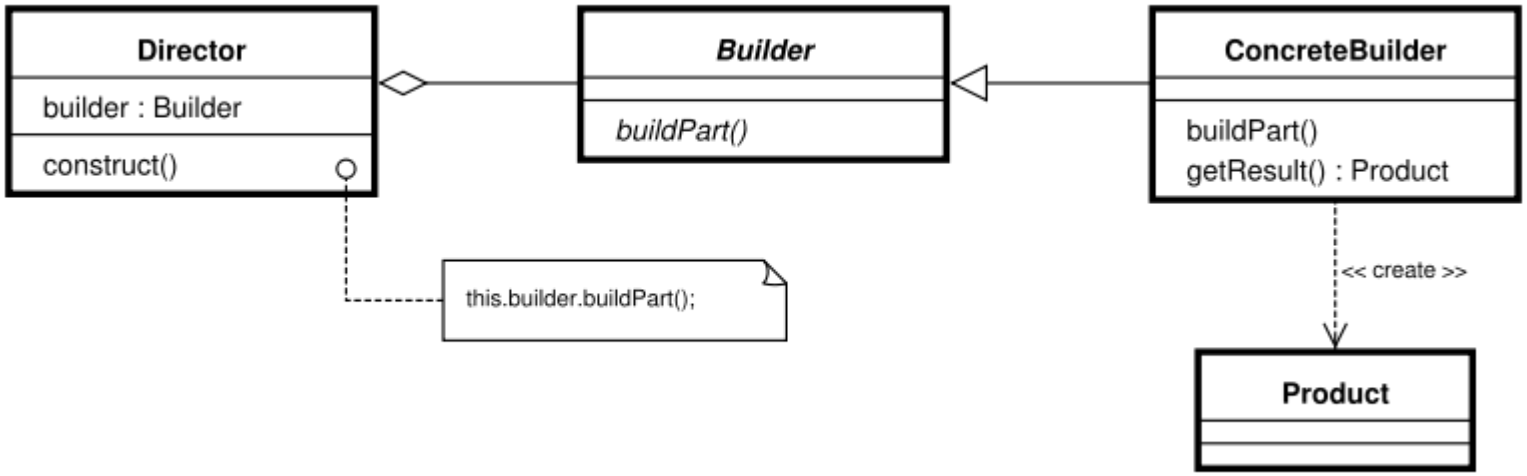
Incentive

- Provide a consistent way to create and configure complex objects
- Provide a clear representation to a complex object

The Problem


client
+ getProduct()
+ use()
Product
+ setId()
+ setName()
Concrete
Client
Concrete
Product

Builder pattern - Solution



Related Patterns

- **Abstract Factory**
- **Factory Method**

Code Example

```
//Product
/* Holds a complex account data*/
class Account{

      private long id;
      private AccountType type;
      private Calendar start,end;
      private double balance, commission;

      set() & get()..

      //many complex operations
      public void withdraw(double amount) {..}
      public void deposit(double amount) {..}
      public void closeAccount() {..}
}
```

Code Example

```
//Account Builder
/* This defines the interface of objects that builder creates
*/
abstract class AccountBuilder{
      protected Account account;

      public Account getAccount (){
            return account;
      }
      public void createNewAccount(){
            account = new Account();
      }
      public abstract void setType();
      public abstract void setCommission();
      ..
}
```

Code Example

```
// Concrete builder for regular account
public class RegularAccountBuilder extends AccountBuilder{
      public abstract void setType(){
            product.setType(ProductType.REGULAR);
      }
      public abstract void setCommission(){
            product.setCommission(Commissions.REGULAR_COMMISSION)
}
```

```
// Concrete builder for gold account
public class GoldAccountBuilder extends AccountBuilder{
      public abstract void setType(){
            product.setType(ProductType.GOLD);
      }
      public abstract void setCommission(){
            product.setCommission(Commissions.GOLD_COMMISSION)
}
```

Code Example

```
// director – set with a concrete builder and uses its methods to build an account
public class AccountDirector{
	private AccountBuilder builder;

	public void setAccountBuilder(AccountBuilder builder){ this.builder=builder;}
	public Account getAccount(){  return builder.getAccount(); }

	public void constructAccount(){
		builder.createNewAccount();
		builder.setType();
		builder.setCommission();
	}
}
```

Code Example

```
// client
public class Client{

    public static void main (String [] args ){

        AccountDirector director=new AccountDirector();
        GoldAccountBuilder gb=new GoldAccountBuilder();

        director.setAccountBuilder(gb);
        director.constructAccount();
        Account account = director.getAccount();
    }
}
```

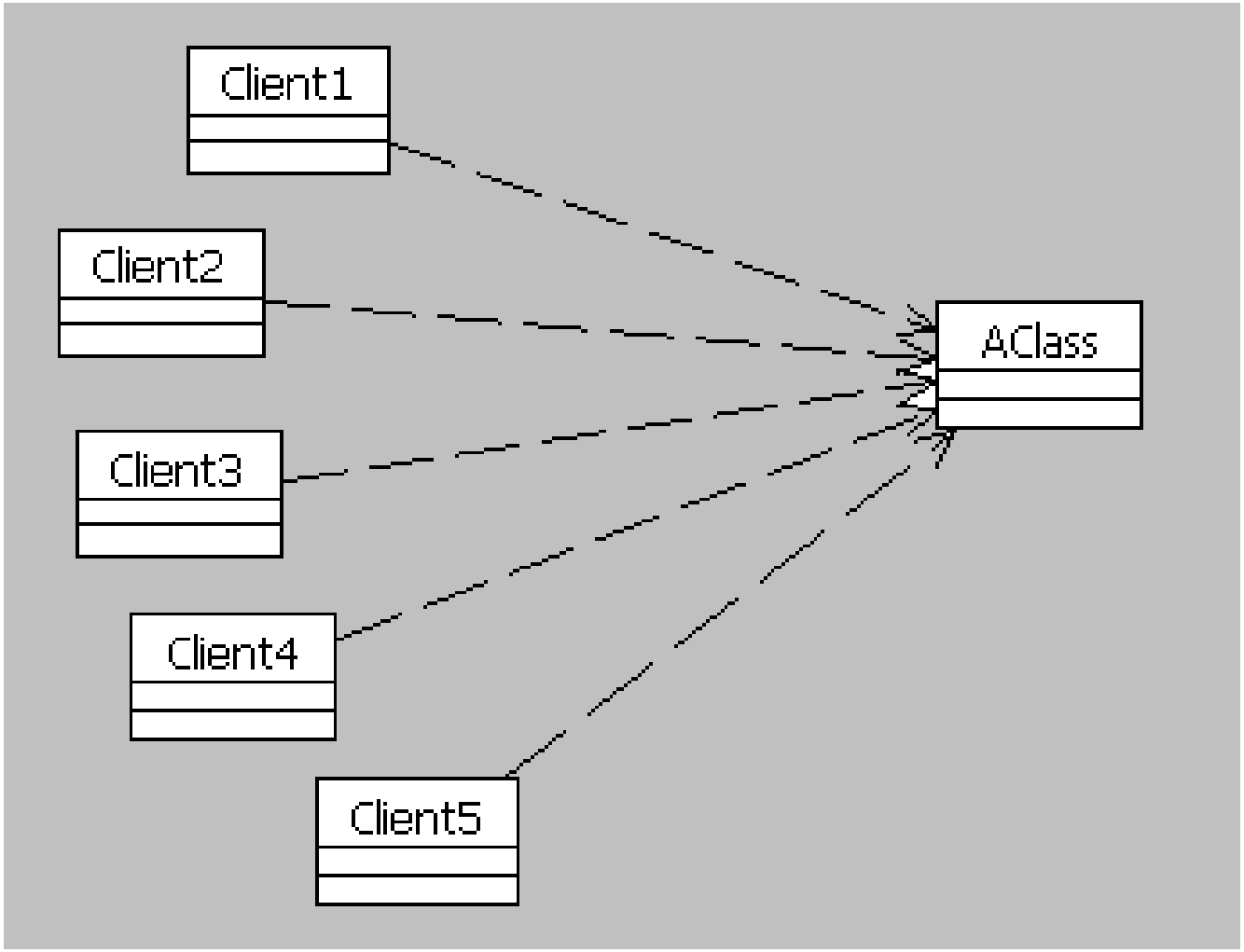
Singleton

- One instantiation that is globally shared
- The class itself responsible for the creation and lifetime

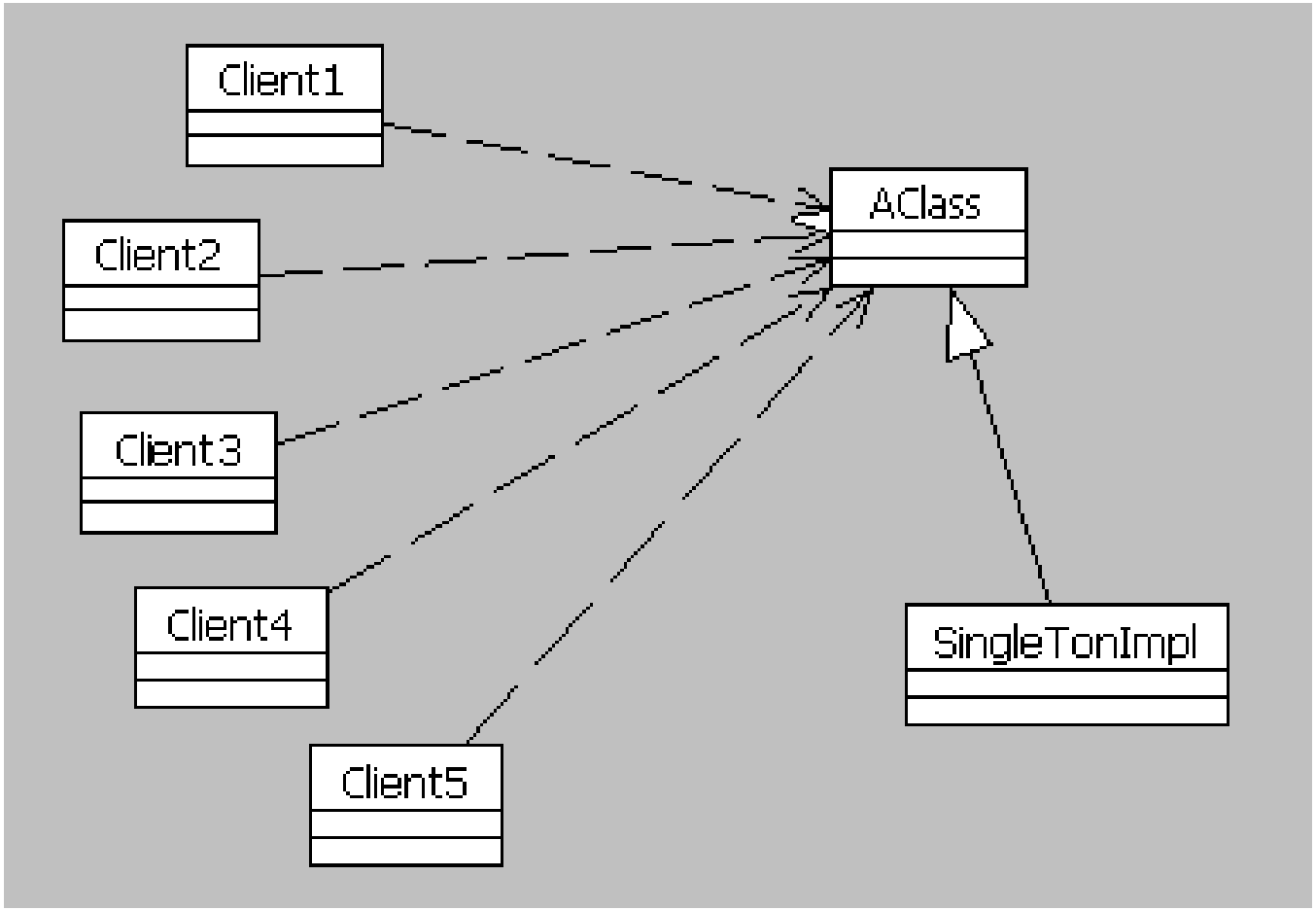
Goal

- One instance per class

Problem



The Singleton Solution



Incentive

- We can delegate the responsibility to manage the instance to a class.
- The class ensures that no other instance of the same class, co-exist.

Code Example

```
/*
Here the class creates an instance when it doesn’t exist and returns the same instance other
    times.
*/
Example-
//Singleton class
class Singleton{
    private Singleton(){};
    private static Singleton uniqueInstance;
    public static Singleton getInstance(){
        if(uniqueInstance==null){
                uniqueInstance=new Singleton();
                System.out.println("Singleton class is instantiated now");
                return uniqueInstance;
        }else{
                System.out.println("Singleton class has already been instantiated");
                return uniqueInstance;
        }
    }
}
```

Code Example

```
//Client
public class Client{
	public static void main(String args[]){
		for(int i=0;i<3;++i){
			Singleton instance=Singleton.getInstance();
		}
	}
}
```

Prototype

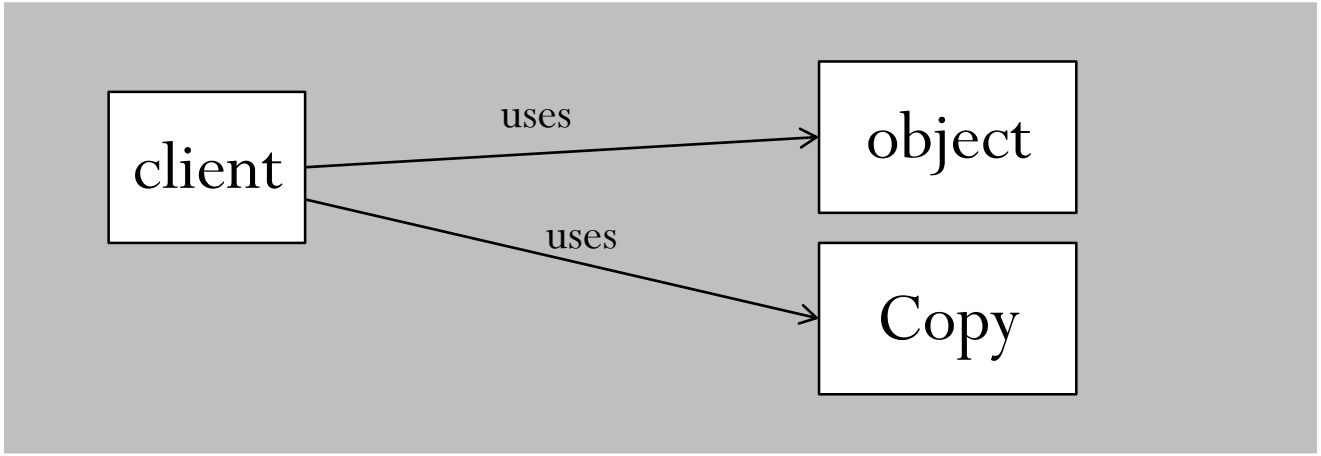
- Constructing new Object from a given Object - Clone

Goal

- Create copies of Objects in runtime without knowing their actual nature in advance

Problem

- You don’t know the actual nature / type of Object when you process it for cloning
- You need to create similar objects (or identical, e.g. clone) from a given object

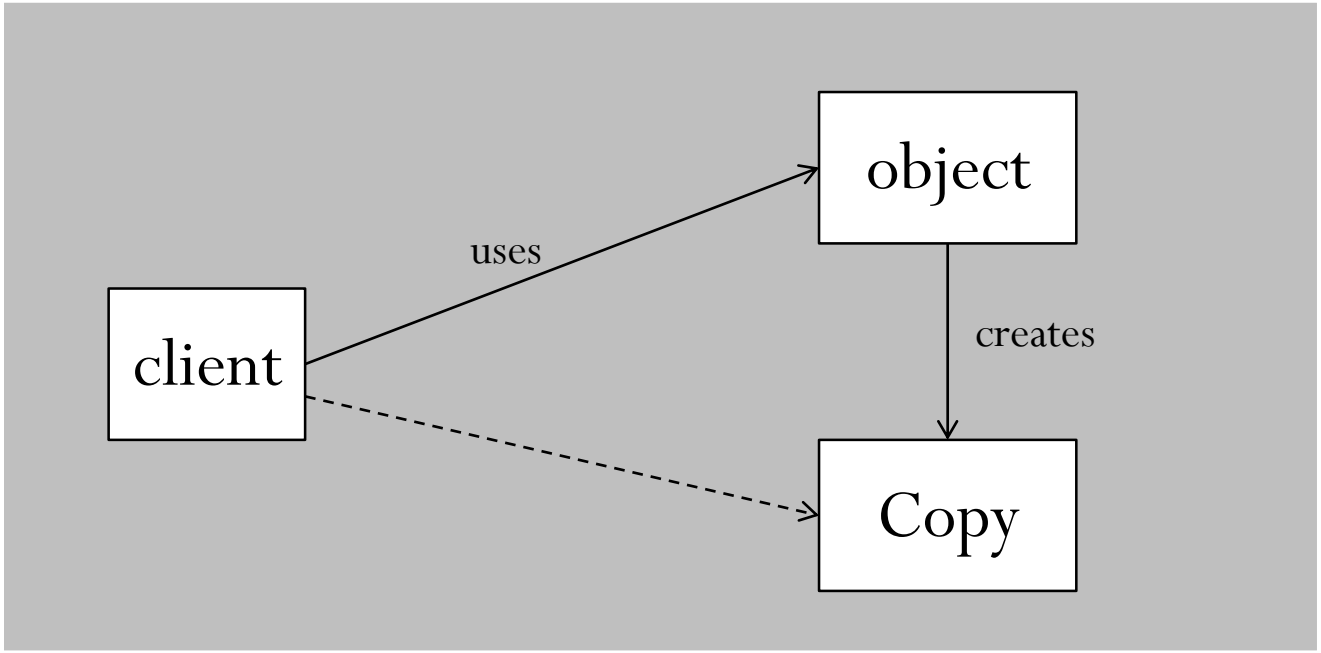


Problem

- Client is forced to
  - check the origin object type (class)
  - call the origin class constructor to instantiate a copy instance
  - verify everything is well copies and populated in that copy
- To much time, performance and tightly coupling design…

The Prototype Solution

- Client uses the origin object in order to construct a copy
- Object becomes a Factory of its own replications



Incentive

- We can delegate the responsibility of creating a copy of an object to that object
- All the complexities are encapsulated inside the origin object instead of bothering the client that needs a copy

- In Java we may use Object.clone() & Clonable interface for Prototype DP

Object Cloning

- Cloning = create new instance with identical state to an existing object

- In Java:
  - Objects are not cloneable by default
  - Java.lang.Object.clone() performs shallow copy
  - You must override it to make your objects clonable

Object Cloning

- Shallow clone
  - Clones the “surface” alone
  - Means that the cloned object references are copies – not re-created !
  - The result is that the original and the clone are holding the same references to their members
  - In Java collections clone() methods returns a shallow copy of the collection

- Deep clone
  - Clones the whole object state
  - Means that references held by the original objects are cloned as well
  - The result is a duplicated referencing entity with its own references
  - Heavy !!!
  - Recursive – all must be cloneable along the referencing chain

Object Cloning

- Deep clone
  - Classes that support deep cloning must implement **Cloneable** interface
  - The interface is empty but it forces a specific convention for Object.clone()
  - The implementation might catch sub-CloneNotSupportedExceptions and decide whether to delegate them or to replace them with default values (not exactly cloning…)

- Some problems and issues to consider:
  - Since Clonenable interface is empty, we can never know if an object support deep cloning via reflection
  - We assume that all referenced objects supports deep cloning as well – risky..
  - Serialization is also a (inefficient) way to clone.…

Object Cloning

- Cloning with creational design patterns
  - Factories
    - Shouldn’t clone objects
    - They rather cache objects for future use
  - Builders
    - May clone an object and configure it differently before handed to the director
  - Singletons
    - Mustn't clone objects
    - Clone() method must be overridden to throw **CloneNotSupportedException**

Code Example

```
// This is a Prototype class – which supports Cloning

public class Ticket implements Clonable {
      private Date date;
      private long serial;
      private String title;

      public Ticket clone(){
              Ticket copy=null;
               try{
                      copy =(Ticket) super.clone();
              }catch(CloneNotSupportedException e){
                      //handle clone exception here when not supported
              }
             //complete any deep clone logic required to make your copy ready
              return copy;
     }
     ……
}
```

Code Example

```
//Concrete tickets
public class MovieTicket { … }

public class FootBallGameTicket { … }

public class TheaterTicket { … }


//client class
public class TicketMaker{
      private Ticket toCopy;
      public TicketMaker (Ticket t){
            toCopy=t;
      }
      public Ticket makeTicket(){
            return toCopy.clone();
      }
}
```

Code Example

```
// working with Prototypes
//generating 100 Movie Tickets:
public class Test{
    public static void main (String [] args){
        Ticket t=new MovieTicket();
        t.setTitle (“Lord Of the Rings – The Return of the King”);
        TicketMaker maker =new TicketMaker( t );
        for(int i=0;i<100;i++){
            Ticket copy = maker.makeTicket();
        …..
        }
    }
}
```

hi-tech college
JOHN BRYCE
Leading in IT Education
a matrix company

Structural Patterns

Adapter pattern

- Wrap up existing classes inside a new target interface
- Helps in the reuse of existing class with a different interface
- Achieving it by converting an existing interface to a new interface.

The Adapter Pattern- Goal

- Convert the interface of a class into another interface that clients expect.

The Problem

- We have a doubly-linked list, built for our object, but we need a stack.

Solution 1

- We use the linked-list interface directly to comprise the stack functionality.
- Read from the Stack will be done by:
  - MoveToTail, IsEmpty(), GetElement, RemoveElement()
- Write to Stack:
  - MoveToTail, InsertElement

Adapter Solution

- The interface Stack is implemented by the AdapterClass.
- Simpler Client.

In General Terms

- A class implements an interface (IA), is needed by a client which expects another interface signature (IB).
- An Adapter class is added to wrap IA and converts it to IB.

Incentive

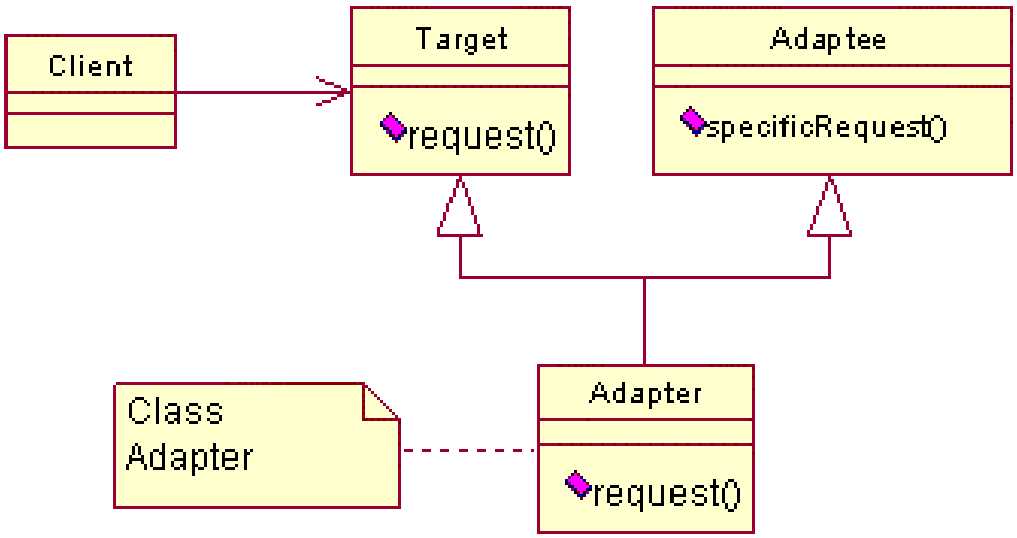
- Implementation is possible in two styles:
  - **class adapters**
  - **object adapters**

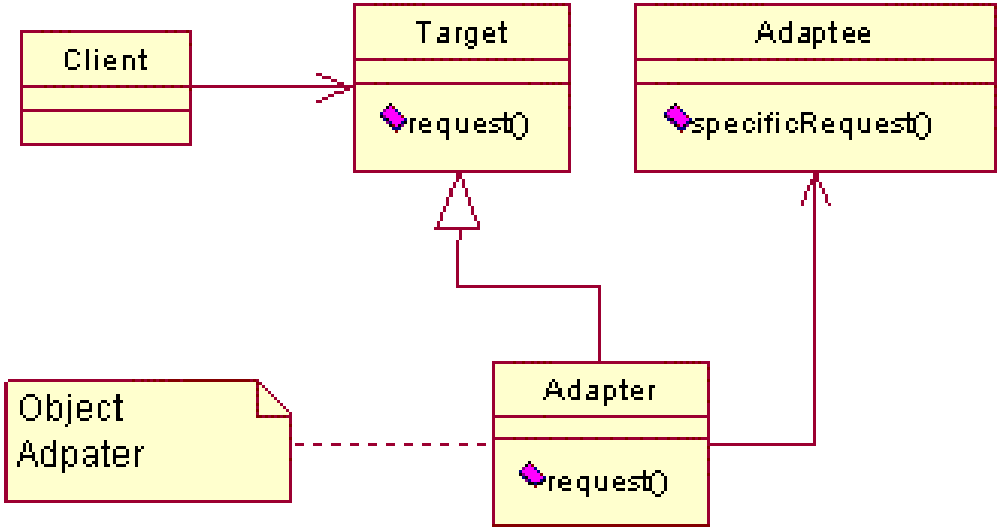
Class adapter

- Uses inheritance so it can be used to adapt only a class and all its parents
- Introduces only one object and hence, avoids additional pointer indirection.

Object adapters

- Use composition to make an Adapter work with many Adaptees.
- Can’t override Adaptee behavior as it doesn’t know which Adaptee it is working with.


Client
Target
request()
Adaptee
specificRequest()
Class
Adapter
Adapter
request()


Client
Target
request()
Adaptee
specificRequest()
Object
Adpater
Adapter
request()

Also Known as and Related Patterns

- Known as Wrapper.
- Bridge -has a different intent of separating the interface from the implementation.
- Decorator more transparent than Adapters.
- Proxy defines a representation for another object without changing its interface.

Code Example

EXAMPLE-1

In a Class Adapter pattern, Adapter extends the Adaptee. When the adapter is used in client code, it works as a Hashtable and gives get ("X") function signature.

The class adapter is supposed to extend the Target as well, but Java doesn't support multiple inheritance, which is achieved by using multiple interface implementation. In this case, we can implement Map interface that Hashtable implements. But for simplicity we have ignored the implementation of the interface.

```
class myAdapter extends Adaptee
{
	Hashtable variables = new Hashtable( 20);
	public myAdapter()
	{
		// calling methods of parent class i.e. the Adaptee
		variables.put( "AUTH_TYPE" , getAuthType());
		variables.put( "REMOTE_USER", getRemoteUser());
		//other methods
	}
	public Object get(Object key)
	{
		return variables.get( key );
	}
	//other methods
}
```

Code Example

```
class myAdapter extends Hashtable
{
	Hashtable variables = new Hashtable(20);
	// composition of an object
	public myAdapter( Adaptee adaptee )
	{
		variables.put( "AUTH_TYPE" , adaptee.getAuthType());
		variables.put( "REMOTE_USER" , adaptee.getRemoteUser());
		//other methods
	}
	public Object get(Object key)
	{
		return variables.get( key );
	}
//other methods
}
```

Facade Pattern

- The Entry Point to each subsystem
- Provides a simplified interface and minimizes the dependency between subsystems.

Goal

- 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.

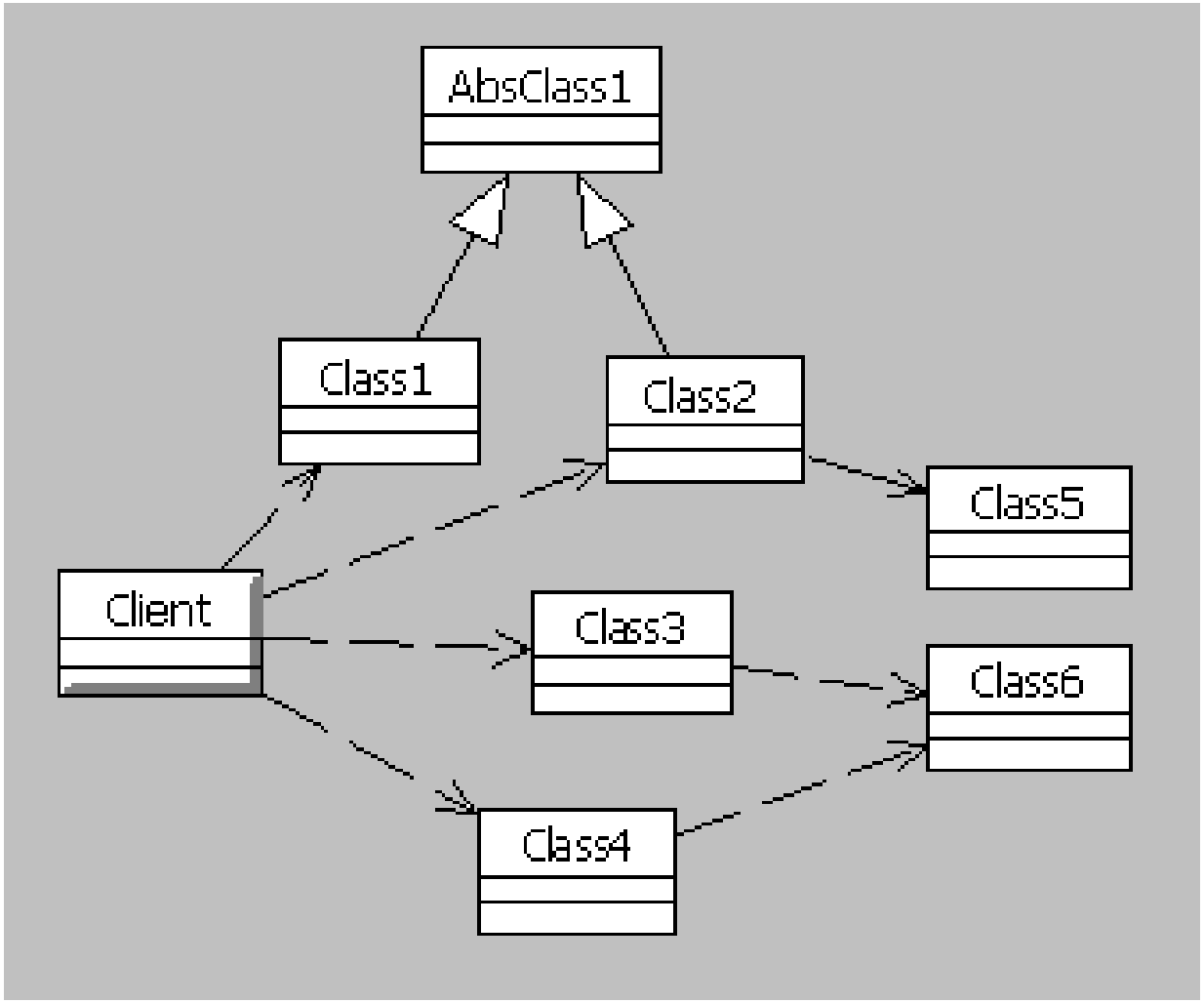
In General Terms

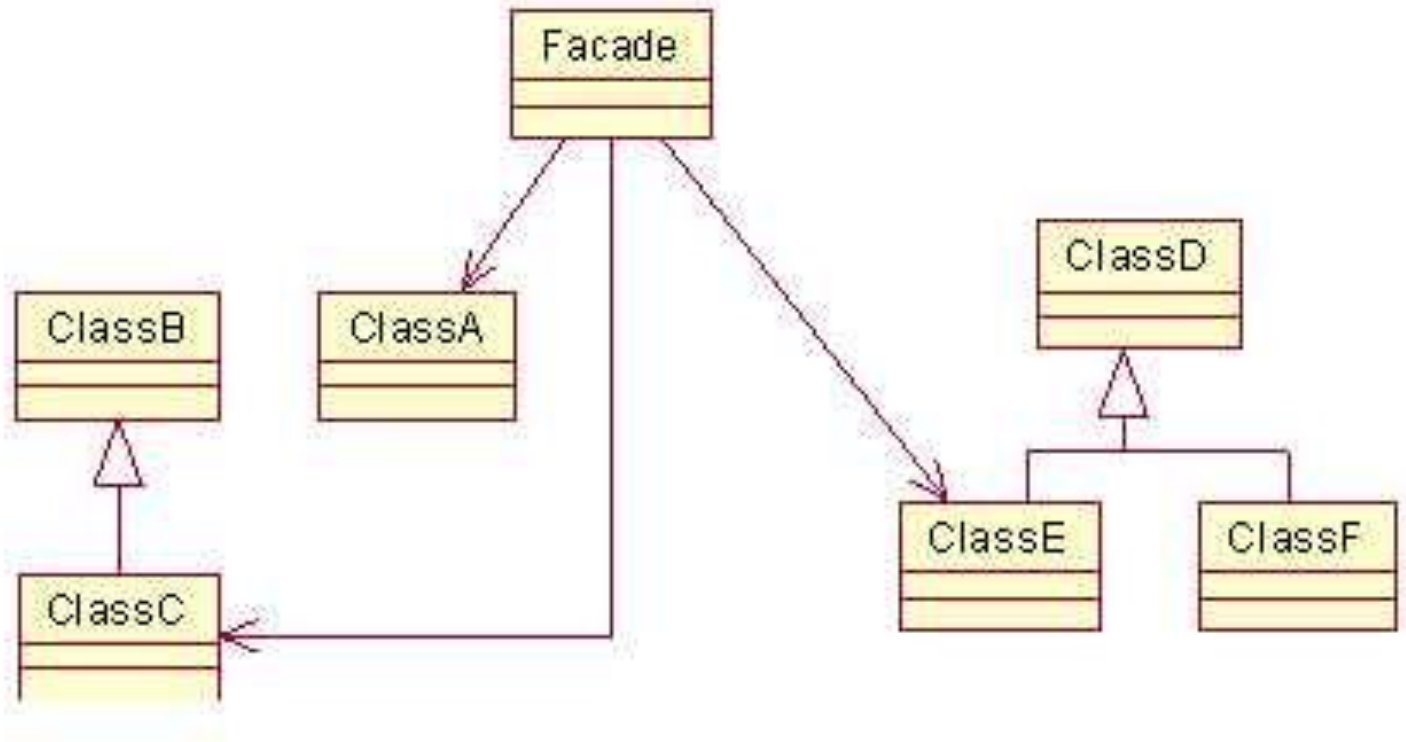
- Simple, default view, of the complex subsystem.
- Helps reduce complexity.
- Generally results in more and smaller classes.

Incentive

- A Facade can be used as an Entry Point to each subsystem level.
- A high level interface can shield classes from the client.

The Problem


AbsClass1
Class1
Class2
Class5
Client
Class3
Class6
Class4


Facade
ClassB
ClassA
ClassD
ClassE
ClassF
ClassC

In Simpler Terms

- An abstract Façade class
- Then the client can communicate with the subsystem through the abstract facade class.

Related Patterns

- Abstract Factory
- Mediator
- Facade objects are often singletons

Code Example

```
/*
In the example the subsystem classes are Scanner, Parser ProgramNode etc.
Compiler is the Façade class, which delegates the low level compilation activities to the
    subsystem classes.
*/
public class Scanner{
    public Token scan(){
        //scanning code
    }
}

public class Parser{
    public void Parse(Scanner scanner, ProgramNodeBuilder programNodeBuilder){
        //parsing code
    }
}
```

Code Example

```
public class ProgramNodeBuilder{
    //code to create the ProgramNode Tree
}

//Façade class
public class Compiler{
    public void compile(InputStream input, OutputStream output){
        Scanner scanner = new Scanner(input);
        ProgramNodeBuilder builder = new ProgramNodeBuilder();
        Parser parser = new Parser();
        parser.Parse(scanner,builder);
        //other compiler activities
    }
}
```

Chain of Responsibility Pattern

- Creating a chain of command processors in order to handle specific object request
- Processing is dynamic – can be narrowed and/or extended

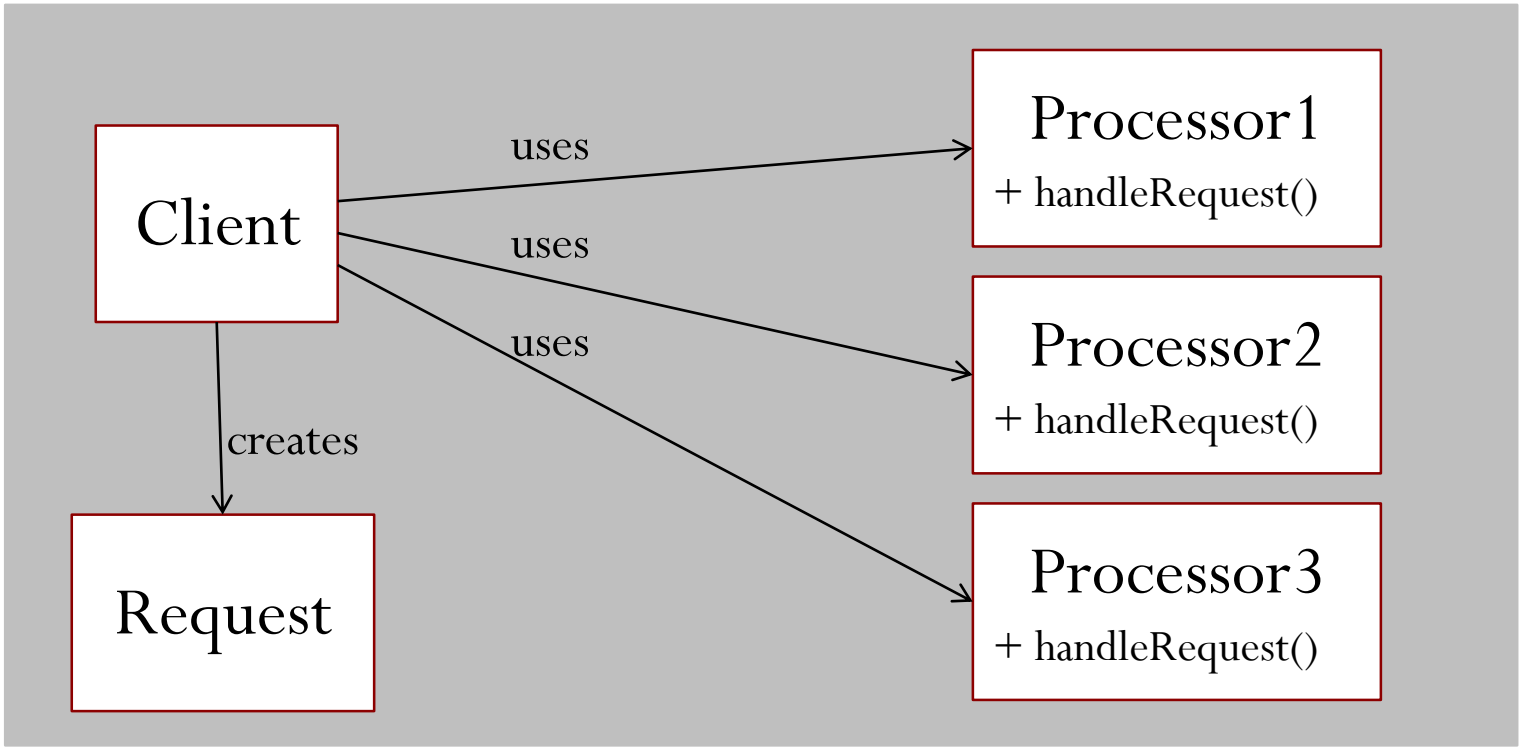
Goal

- Provide a loosely coupled mechanism for passing a given command from specific object to the commands processors.
- Each processor provides its own logic when relevant
- Processors chain can be extended / narrowed

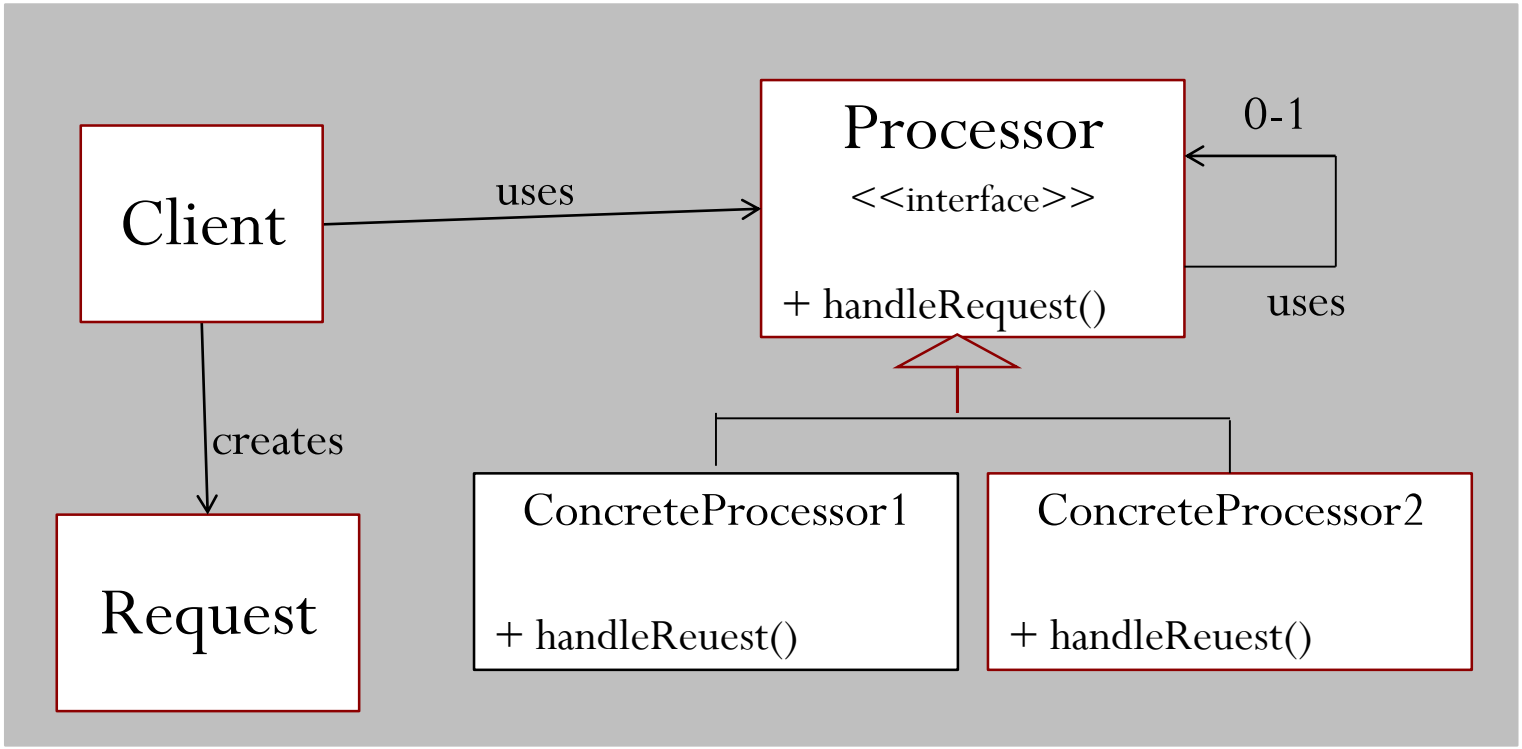
Incentive

- A Chain of Responsibility can decouple the client request from its processing complexity
- Processors are totally isolated and can be maintained separately

The Problem


Client
uses
uses
uses
creates
Request
Processor1
+ handleRequest()
Processor2
+ handleRequest()
Processor3
+ handleRequest()

The Chain of Responsibility Solution



In Simpler Terms

- An abstract Processor class used by all concrete Processors
- Support request delegation between each processor to the next one on the chain
- Client generates a request (command) and deposit it to the first processor or the chain
- The request is now being delegated by the processors in the chain and gets handled
- Client is not aware to the chain structure and complexity
- The chain may change structure in run-time
- More new concrete processors can be authored and combined with existing

Code Example

```
public abstract class AbstractLogger {
   public static int INFO = 1;
   public static int DEBUG = 2;
   public static int ERROR = 3;
   private AbstractLogger nextLogger;
   protected int level = 2;

   //next element in chain or responsibility
   protected AbstractLogger nextLogger;

   public void setNextLogger(AbstractLogger nextLogger){
      this.nextLogger = nextLogger;
   }

   public void logMessage(int level, String message){
      if(this.level <= level){
         write(message);
      }
      if(nextLogger !=null){
         nextLogger.logMessage(level, message);
      }
   }

   abstract protected void write(String message);

}
```

Code Example

```
public class ConsoleLogger extends AbstractLogger {

   public ConsoleLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
            System.out.println("Standard Console - Logger: " + message);
   }
}
```

```
public class ErrorLogger extends AbstractLogger {

   public ErrorLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
            System.out.println("Error Console - Logger: " + message);
   }
}
```

Code Example

```
public class FileLogger extends AbstractLogger {

   public FileLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
            System.out.println("FileConsole - Logger: " + message);
   }
}
```

Code Example

```
public class ChainPatternDemo {

   private static AbstractLogger getChainOfLoggers(){
      AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
      AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
      AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);
      errorLogger.setNextLogger(fileLogger);
      fileLogger.setNextLogger(consoleLogger);
       return errorLogger;
   }

   public static void main(String[] args) {
      AbstractLogger loggerChain = getChainOfLoggers();
      loggerChain.logMessage(AbstractLogger.INFO,
         "This is an information.");
      loggerChain.logMessage(AbstractLogger.DEBUG,
         "This is an debug level information.");
      loggerChain.logMessage(AbstractLogger.ERROR,
         "This is an error information.");
   }
}
```

The Bridge Pattern

- Bridges abstraction with implementation
- They can vary independently.
- Used upfront in design.

Bridge Pattern- Goal

- Decouple an abstraction from its implementation so that both can vary independently.

In General Terms

- The catch here is "decouple abstraction from implementation".

Inheritance Vs Bridge

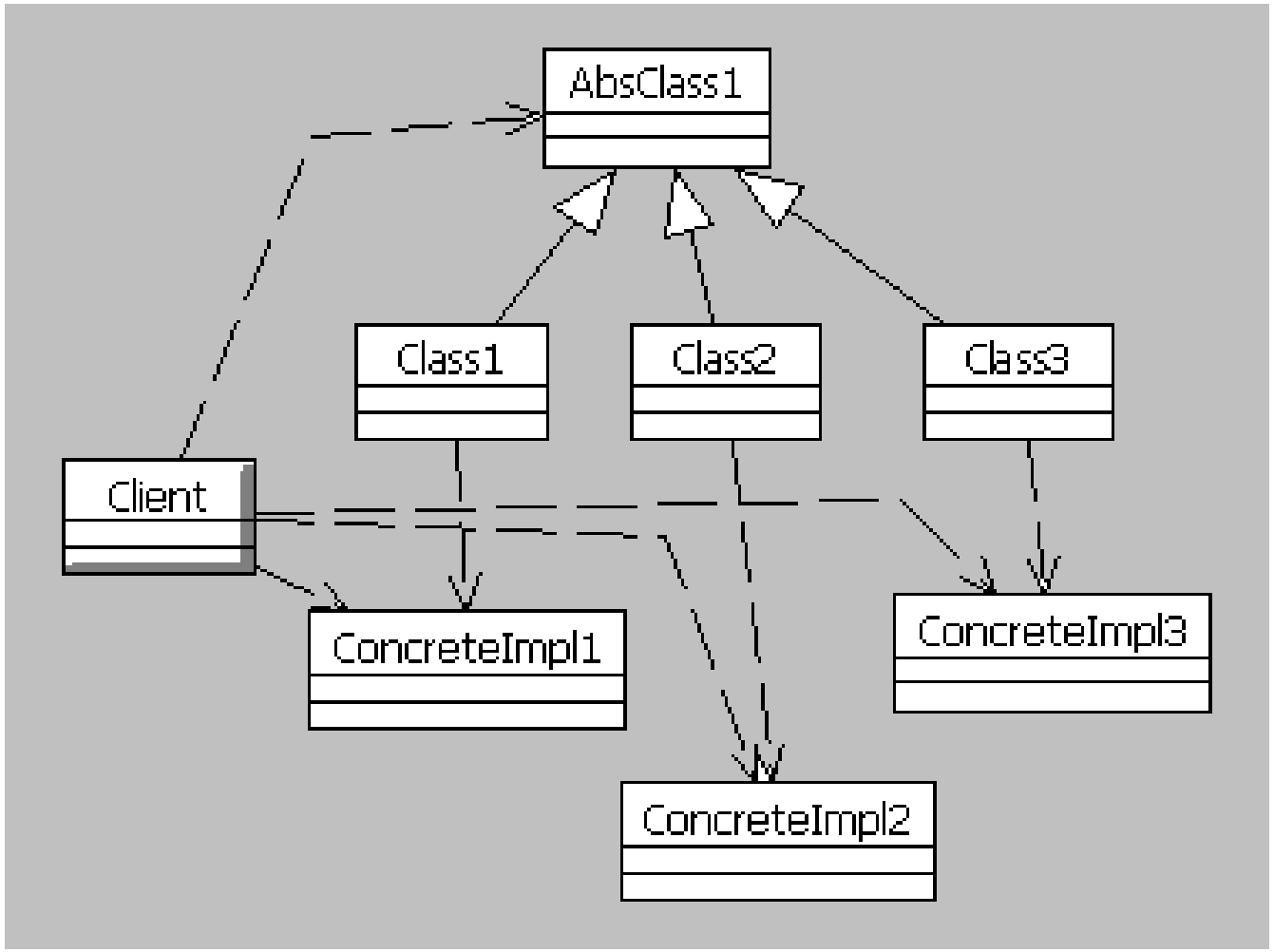
- Inheritance provides different implementations for an abstraction, but it is no flexibility
- Inheritance is difficult to modify, extend and reuse
- Bridge pattern solves the problem by separating the abstraction class and implementation class

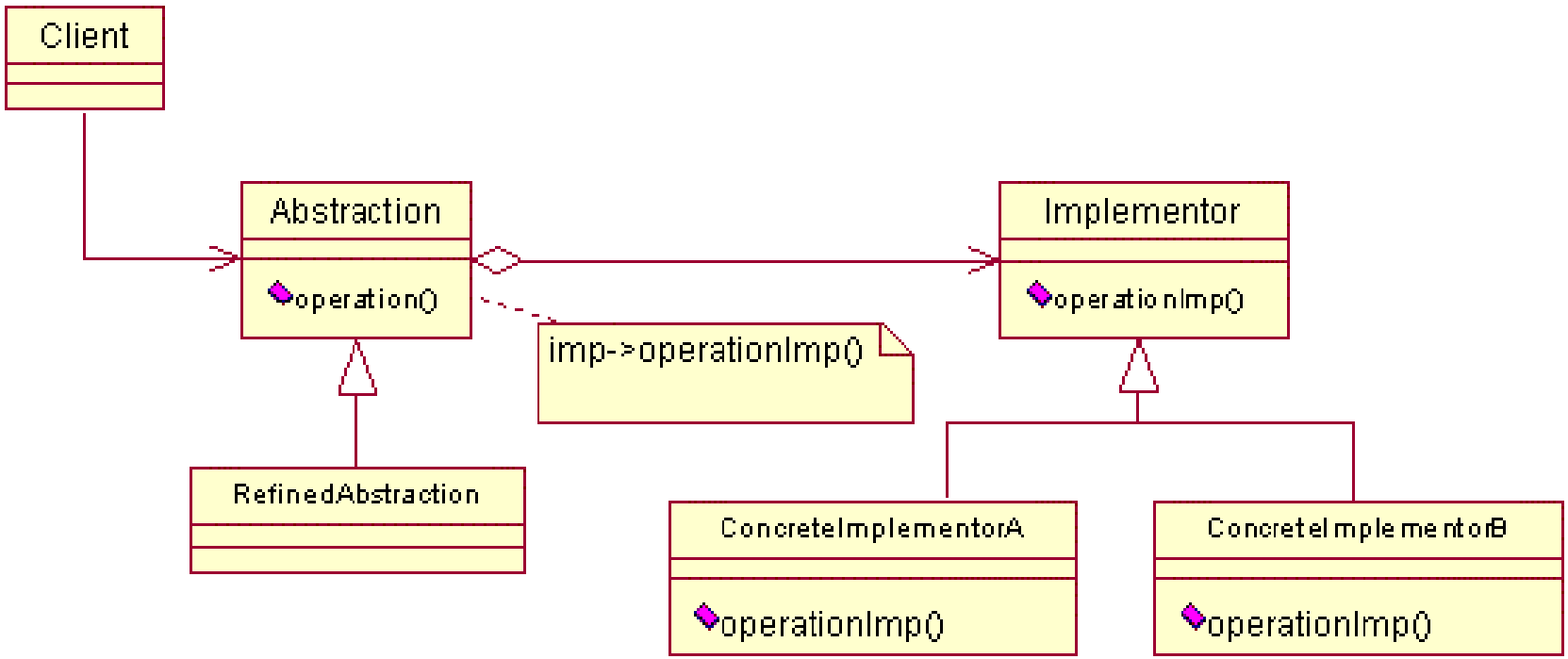
Incentive

How to judge the applicability of the Bridge pattern:

1. When the client wants to avoid a permanent binding
2. Should be able to extend both abstraction and implementation
3. Changes in actual implementation is transparent
4. The details is seen only by implementers
5. Better manageability
6. Implementation is shared among multiple objects without client knowledge.

The Problem


AbsClass1
Class1
Class2
Class3
Client
ConcreteImpl1
ConcreteImpl3
ConcreteImpl2


Client
Abstraction
operation()
Implementor
operationImp()
imp->operationImp()
RefinedAbstraction
ConcreteImplementorA
operationImp()
ConcreteImplementorB
operationImp()

In Simpler Terms

- Decide how to select the Implementor class
- Have an Abstract Factory class to represent the hierarchy of Implementors

Related Patterns

- A Bridge pattern is a design time Design Pattern.
- Adapter pattern
- Abstract Factory

Code Example

```
/*
Various participants in this sample code are Abstraction, RefinedAbstraction, Implementor,
    ConcreteImp and Client class. Abstraction defines the abstraction's interface.
    RefinedAbstraction extends the interface provided by Abstraction. Implementor defines the
    interface for Implementor class and ConcreteImp implements that.
*/

//Abstraction class
abstract class Abstraction{
    protected Implementor imp;
    public void operation()
    {
        imp.operationImpl();
    }
    //Implementation is provided by child classes.
    public abstract void loadImplementor();
}
```

Code Example

```
class RefinedAbstraction extends Abstraction{
        // Do the implementation here
        public void loadImplementor(){
                imp=new ConcreteImplementor();
                return imp;
        }
}

class Implementor{
        public void operationImp(){
                //Implement code here
        }
}

//ConcreteImplementor class
class ConcreteImplementor extends Implementor{
        //Override parent method
        public void operationImp(){
                //Implement the code here
                System.out.println("This is an example of Bridge Pattern");
        }
}
```

Code Example

```
//Client
public class Client{
public static void main(String args[]){
Abstraction abs=new RefinedAbstraction();
abs.loadImplementor();
abs.operation();
      }
}
```

Composite Pattern

- Complex object out of elementary objects
- Explains the context and forces when a pattern can be applied.

Goal

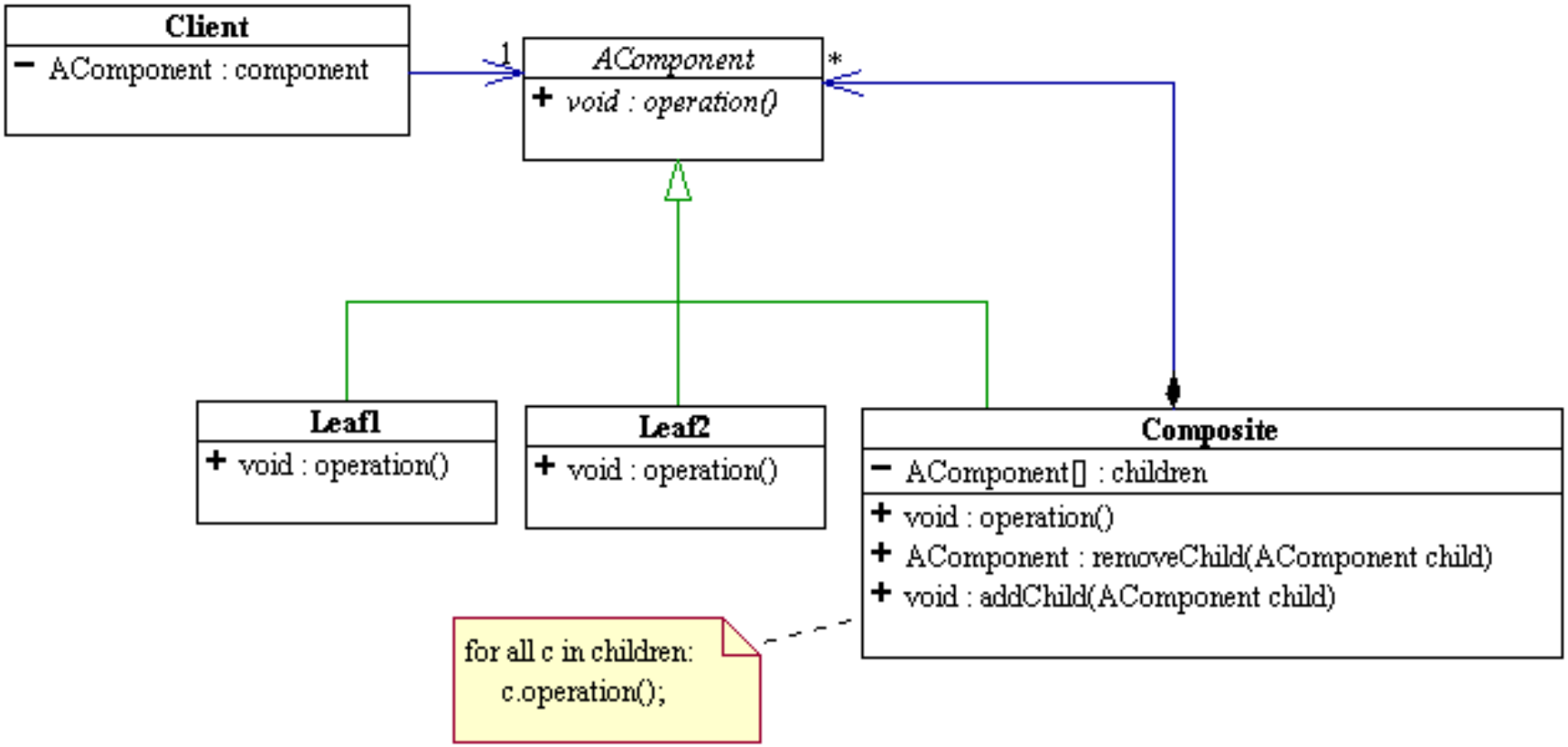
- Compose objects into tree structures
- Lets clients treat individual and compositions of object uniformly.

In General Terms

- Achieving recursive composition by using a tree structure which makes the client simple.
- A node may have a branch i.e. another composite or a primitive object.
- The key to the composite is an abstract class
- This abstract class, called ‘component’, declares all common operations
- Client code manipulates objects in the composition through the component interface

Incentive

- A primitive node doesn’t have any children. A composite node has.
- One group of thought suggests treating both differently and keeping two interfaces.
- Composite retains a uniform interface.
- The client is ignorant about the node structure
- It also makes it easier to add new components


Client
- AComponent : component
1
AComponent
+ void : operation()
*
Leaf1
+ void : operation()
Leaf2
+ void : operation()
Composite
- AComponent[] : children
+ void : operation()
+ AComponent : removeChild(AComponent child)
+ void : addChild(AComponent child)
for all c in children:
c.operation();

Composite Pattern-Sample Code

- Equipment such as computers and stereo components are often organized into part-whole or containment hierarchies.

Explicit parent reference

- Storing the parent component reference in a child simplifies the management
- Easier to move up the structure and deleting a component.

Where should the methods be defined?

- Should the methods be declared in the Component class and let Composite and Leaf inherit them?
- Or, define the operations at a subclass level that gives more safety

Caching child information

- Cache in the Composite class.
- We can also define a listener that will automatically refresh the cache, when there is any update in the tree.

Choose a data structure to store components

- We can choose a data structure out of many available in Java Collection API like ArrayList, HashMap, Hashtable, LinkedList, Vector etc.

Choose a data structure to store components

- Choosing a particular data structure depends on requirement like efficiency, safety etc.
- Vector and Hashtable are both thread-safe and hence slower in execution unlike ArrayList or HashMap.

Related Patterns

- The Chain of Responsibility
- Decorator is used with Composite
- Flyweight
- Iterator to traverse the tree in Composite.
- Visitor

Code Example

```
/*
Various participants in this sample code are Acomponent, Leaf, Composite and Client.
    Acomponent is the Component that defines the interface for objects in composition. It
    also gives a default implementation for the child components. Leaf has no children and
    here any attempt to add or remove a child from Leaf will invoke Processsing Exception.
Composite represents a child node that can have branch nodes.
*/
Example-
import java.util.*;
// Processing Exception class
class ProcessingException extends Exception{
    ProcessingException(){
        super();
    }
}
```

Code Example

```
// Component
abstract class Acomponent{
	Enumeration enum;
	Vector vec;
	Acomponent(){
		vec=new Vector();
	}
	public void add(Acomponent obj) throws ProcessingException{
		vec.add(obj);
	}
	public void remove(Acomponent obj) throws ProcessingException{
		vec.remove(obj);
	}
	public Acomponent getChild(int i) throws ProcessingException{
		return (Acomponent)vec.elementAt(i);
	}
	public Enumeration getChild(){
		System.out.println(vec);
		return vec.elements();
	}
}
```

Code Example

```
class Leaf extends Acomponent{
      private Object o;
       public Leaf(Object data){
            this.data=data;
      }
      public Object getData (){
            return data;
      }
      public void add(Acomponent obj) throws ProcessingException{
            throw new ProcessingException();
      }
      public void remove(Acomponent obj) throws ProcessingException{
            throw new ProcessingException();
      }
      public Acomponent getChild(int i) throws ProcessingException{
            throw new ProcessingException();
      }
}
```

Code Example

```
class Composite extends Acomponent{
      //uses parent class operations
}
```

```
public class Client{
      public static void main(String args[]) throws ProcessingException{
            System.out.println("An example of Composite pattern");
            Acomponent com=new Acomponent();
            /* Write code to add a Composite object here.*/
            //Add child nodes
            com.add(new Leaf(new Integer(1)));
            com.getChild();
            com.add(new Leaf(new Integer(2)));
            com.getChild();
      }
}
```

Proxy Pattern

- A proxy controls access to another object
- Can also defer the full cost of creation

Goal

- Provide a surrogate or placeholder for another object to control access to it.

In General Terms

- Stand-in for another object
- Add a level of indirection
- The Client interacts with the Proxy
- So the Proxy object works as a stand-in for the actual object here.

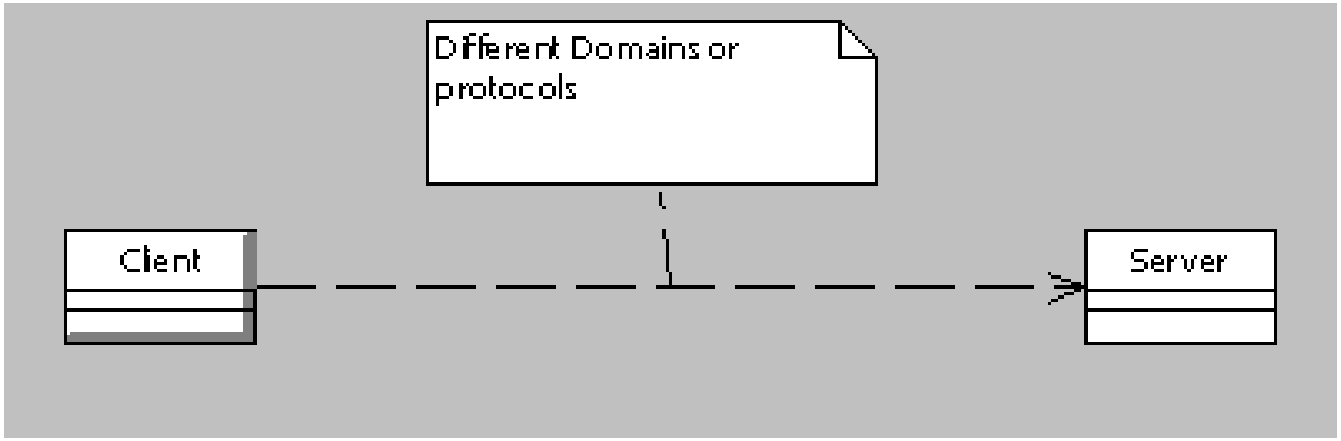
Incentive

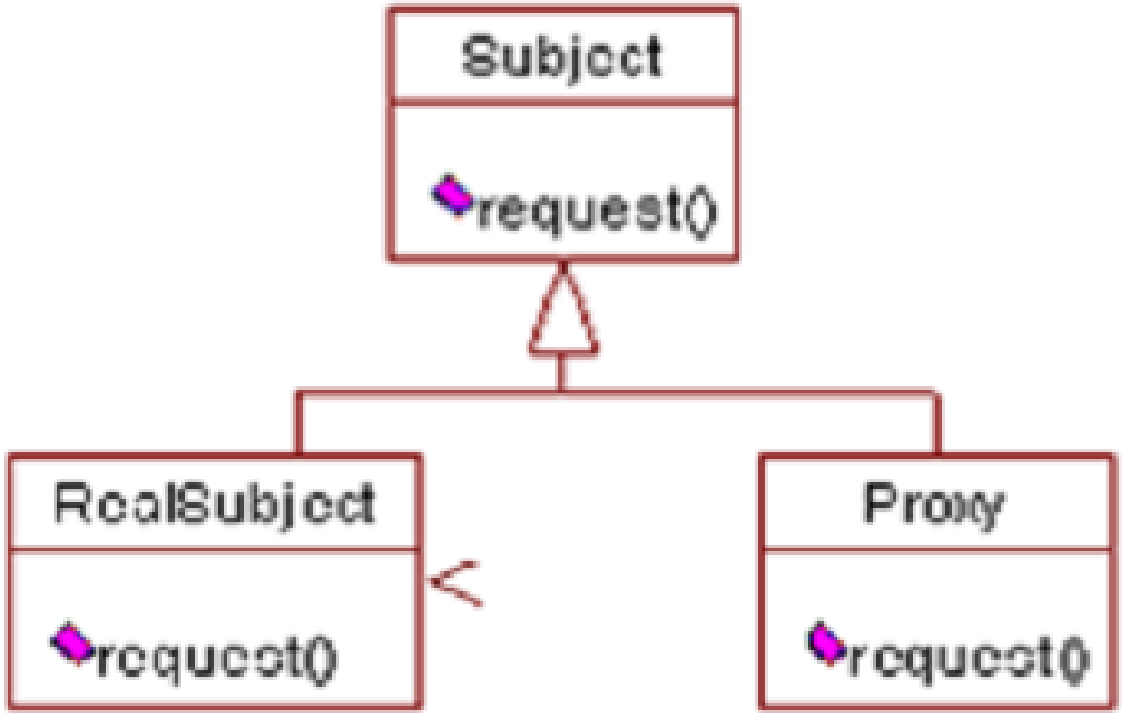
- Apply Proxy Pattern whenever there is a need for a more versatile or sophisticated reference to an object.
- A Proxy can control access to the actual object by authentication.
- It can also cache results returned from actual object.

Pattern Types

- There are various types of Proxy objects.
- The most commonly used Proxy objects:
  - **Cache Proxy**
  - **Count Proxy**
  - **Protection Proxy**
  - **Remote Proxy**
  - **Virtual Proxy**

The Problem




Subject
request()
RealSubject
request()
Proxy
request()

Protection Proxy Example

- The real object stores a secret word.
- Only authorized clients who have the password have access to this word.
- A proxy that knows the password protects real object.

Protection Proxy Example

- If client wants to get the secret word, first the proxy asks the user to authenticate itself.
- If user supplies correct information to the proxy, then the proxy calls the real object and passes the secret word to the client.

Code Example

```
/** The interface exposed to the clients.    */
abstract class Generator{
/** Asks the value of PI                                    */
        public abstract double Get_PI();
/** Asks the value of e                                     */
        public abstract double Get_e();
}
/** The implementation of the Generator    */
class RealGenerator extends Generator{
/** Iinitialize the Generator                               */
        public RealGenerator(){
                System.out.println("RealGenerator.RealGenerator()");
        }
/** Generates PI                                                    */
        public double Get_PI(){
                System.out.println("RealGenerator.Get_PI()");
// This is the place where you can calculate the PI
// if you want
                return java.lang.Math.PI;
        }
/** Generates e                                                     */
        public double Get_e(){
                System.out.println("RealGenerator.Get_e()");
// This is the place where you can calculate the e
// if you want
                return java.lang.Math.E;
        }
}
```

Code Example

```
/** A cache proxy for a Generator                    */
class CacheProxy extends Generator{
/** The referenced Generator object                  */
        RealGenerator realobj=null;
/** Cache value                                                    */
        double Store=0;
/** Cached method                                          */
        int LastAccessed=0;
/** Initialization                                         */
        public CacheProxy(){
                System.out.println("CacheProxy.CacheProxy()");
                realobj=new RealGenerator();
        }
/** Checks if PI is cached. If it returns the cached
value. If it isn't, asks it from the real Generator,
and cache this value. */
        public double Get_PI(){
                System.out.println("CacheProxy.Get_PI()");
                if(LastAccessed!=1){
                        Store=realobj.Get_PI();
                        LastAccessed=1;
                        System.out.println("PI cached");
                }
                return Store;
        }
…….
```

Code Example

```
/** Checks if e is cached. If it is returns the cached
value. If it isn't, asks it from the real Generator,
and cache this value. */
    public double Get_e(){
        System.out.println("CacheProxy.Get_e()");
        if(LastAccessed!=2){
            Store=realobj.Get_e();
            LastAccessed=2;
            System.out.println("e cached");
        }
        return Store;
    }
}
```

Code Example

```
/** Test client for the Cache Proxy application */
public class cache{
      public static void main(String args[]){
                Generator calc=new CacheProxy();
                double d=calc.Get_PI();
                System.out.println("main: "+d);
                d=calc.Get_PI();
                System.out.println("main: "+d);
                d=calc.Get_PI();
                System.out.println("main: "+d);
                d=calc.Get_e();
                System.out.println("main: "+d);
                d=calc.Get_e();
                System.out.println("main: "+d);
                d=calc.Get_PI();
                System.out.println("main: "+d);
                d=calc.Get_e();
                System.out.println("main: "+d);
      }
}}
```

Decorator

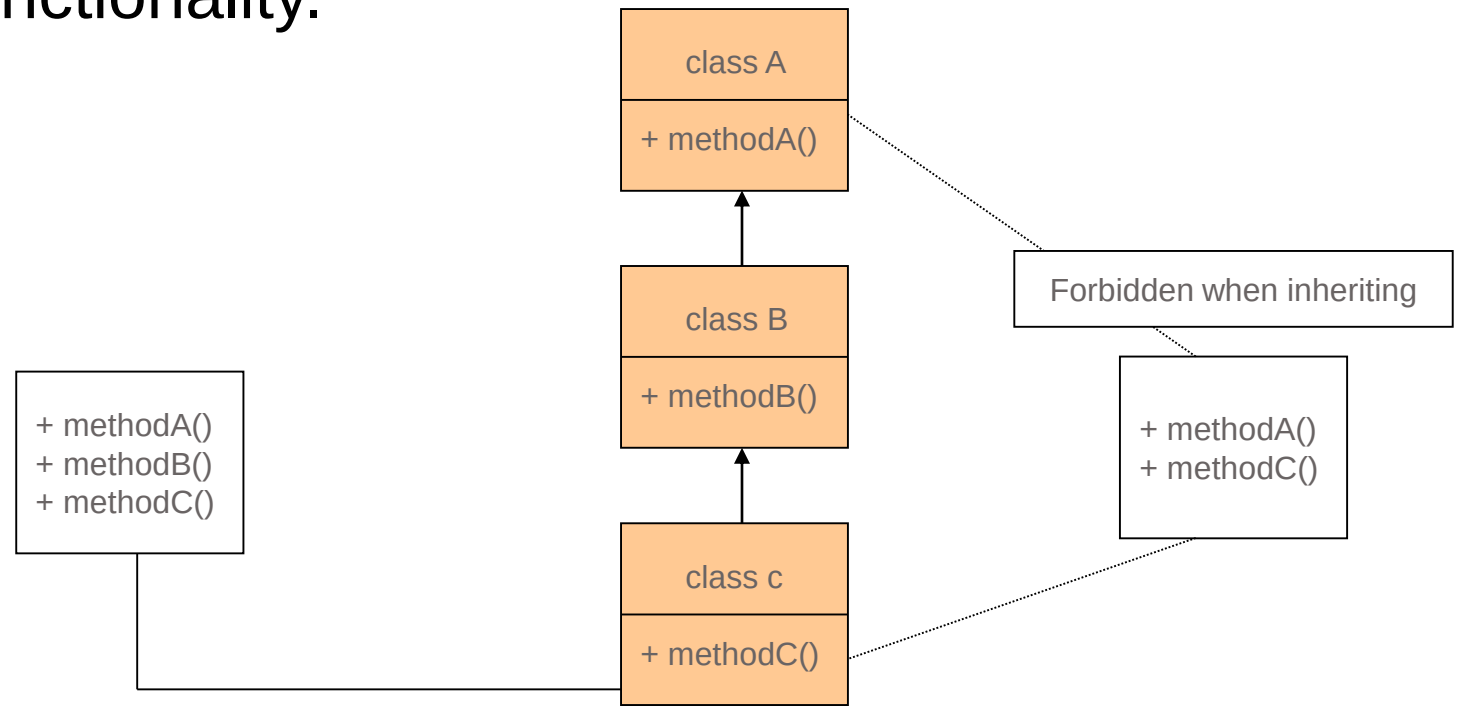
- Adds functionality without sub-classing
- Can be used in a flexible way enabling different ways of flow

Goal

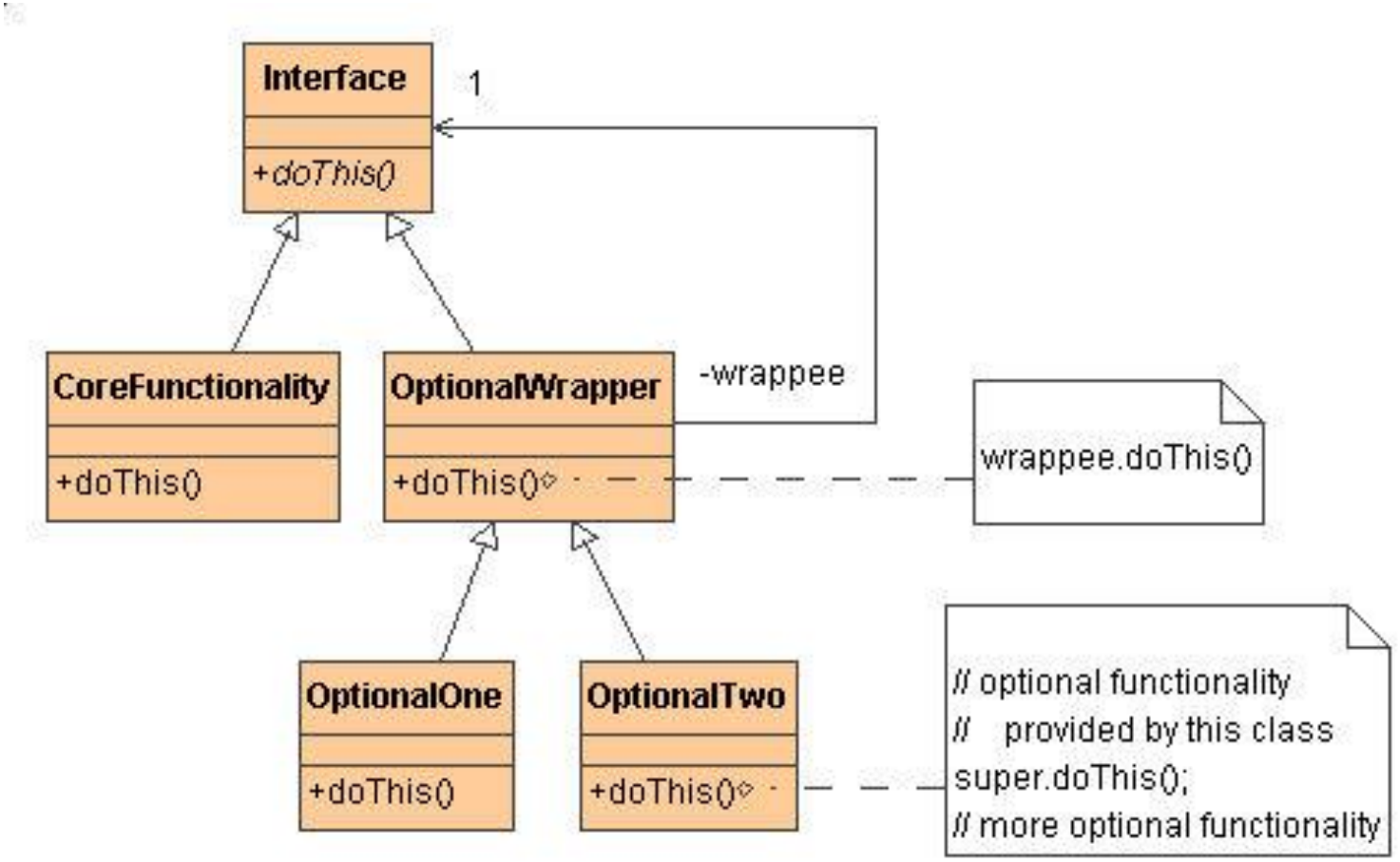
- Add functionality without sub-classing
- The extra functionality wrap can be assigned to different wrappers
- Flexible chains of wrappers can be assigned to any implementation and decorate it

The Problem

- Inheritance creates an inflexible chain of functionality.



The Decorator Solution



Incentive

- Each decorator can be use separately or combined with others

Code Example

```
/*
This code writes an object to a file called “myFile.data”.
The concrete class which writes to the file is wrapped by to decorators:
1- BufferdOutputStram - decorating output streams using a buffer
2- ObjectOutputStream - serializing object data into primitive write calls
*/
ObjectOutputStream out = new ObjectOutputStream (
                                 new BufferedOutputStream (
                                 new FileOutputStream (“myFile.data”)));
out.writeObject(new MyObject());
out.close();
```

Usage in Java

- Java I/O package offers several decorators for convenient way of reading and writing data (bytes, primitives, objects)
- Servlets and JSP tech. –
  - Filters are used for pre-process and post-process incoming messages
  - Each web component may have its own chain of filters

hi-tech college + JOHN BRYCE Leading in IT Education a matrix company

Behavioral Patterns

Iterator

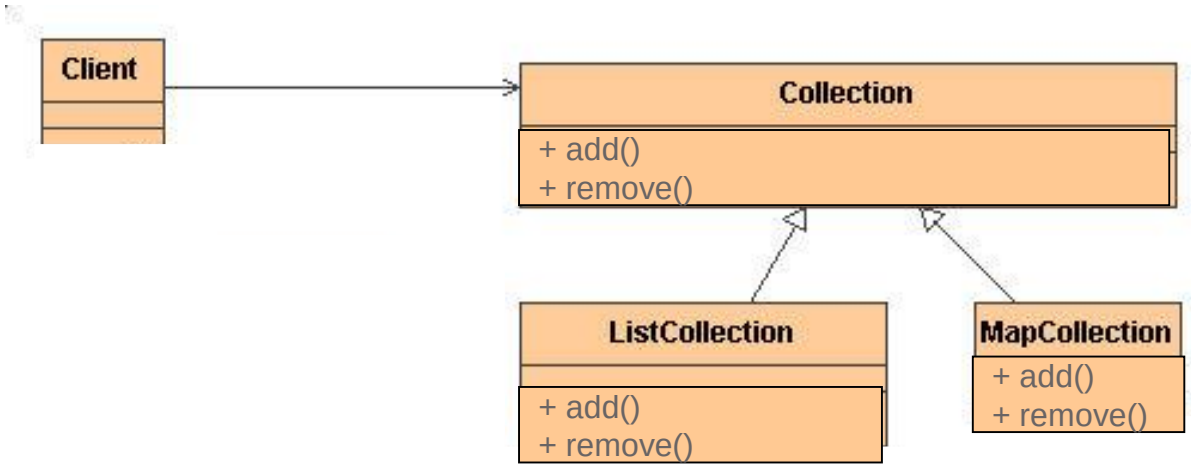
- Accesses the elements of an object collection sequentially
- Done without exposing its underlying representation

Goal

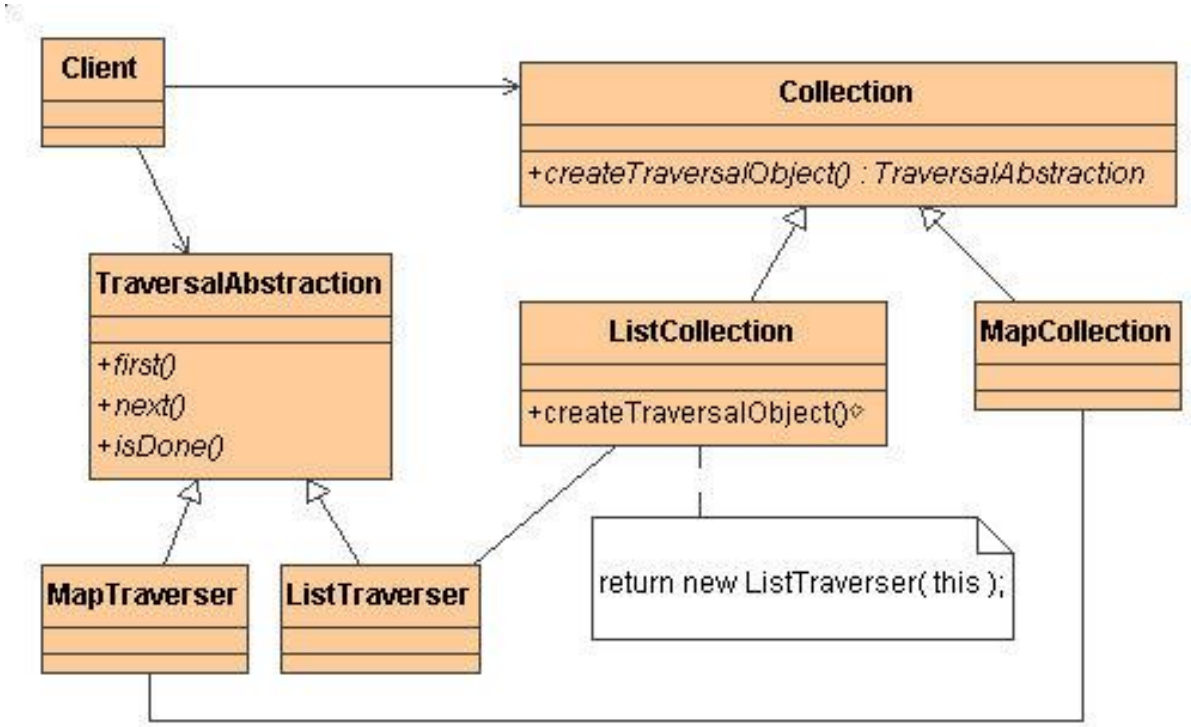
- Browse a collection of entities without dealing with its implementation

The Problem

- Clients needs to be familiar with code and logic



The Iterator Solution



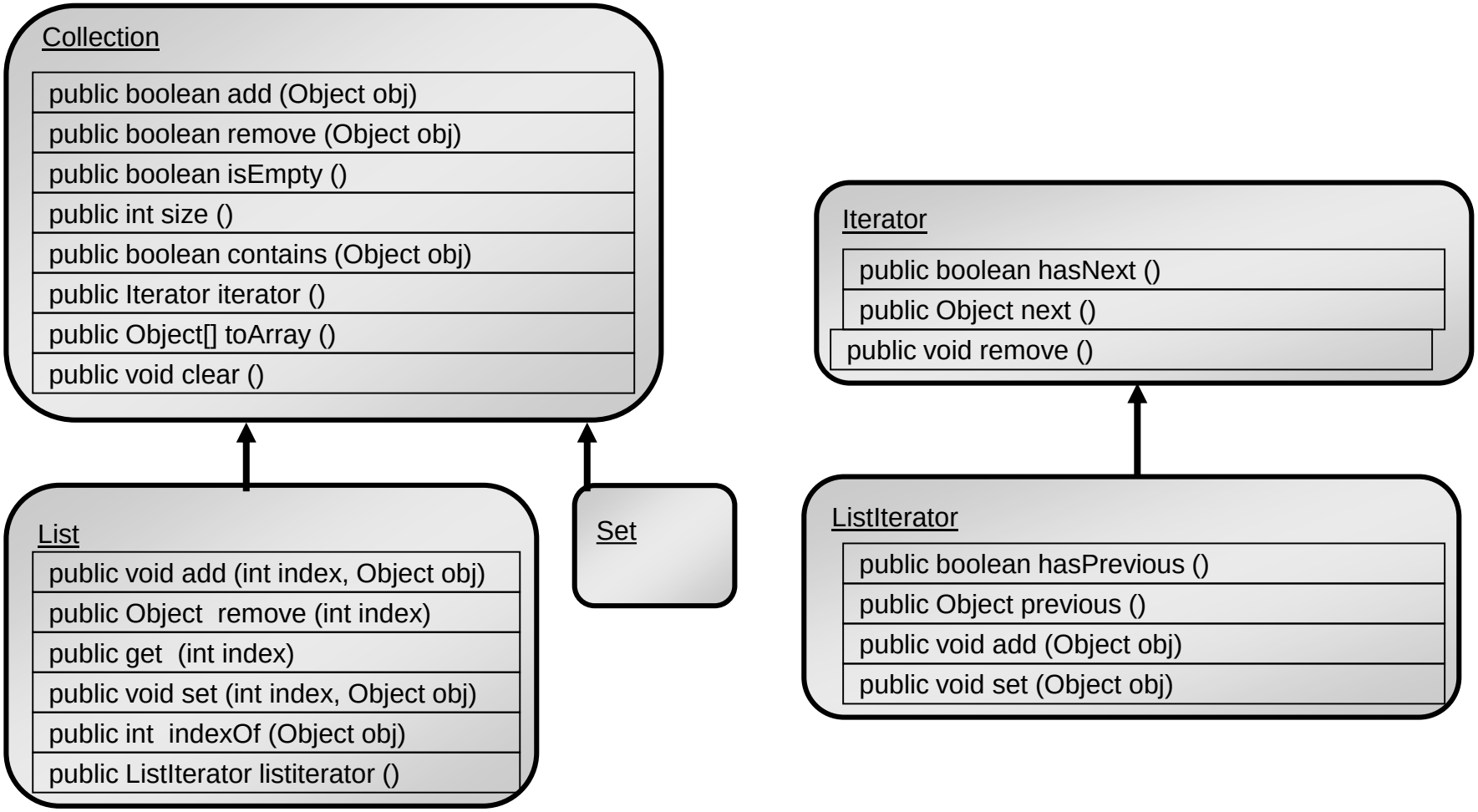
Incentive

- Each collection may implements iterators in their own way

Code Example

```
/*
This code instantiate a Vector which is a List collection and iterates it using its Iterator
*/
Collection col = new Vector ();
// add some elements..
Iterator elements = col.iterator();
while (elements.hasNext())){
      System.out.println(elements.nextElement())
}
```

Java collections – Iterators



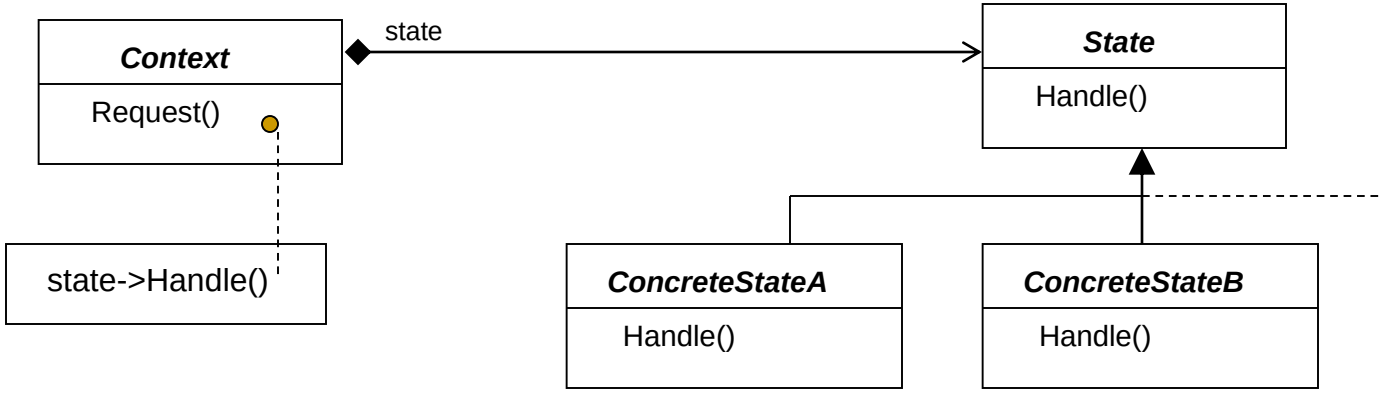
State

- Allow an object to alter its behavior when its internal state changes. The object which appear to change its class.
- “Implementing discrete object states using explicit classes"

Incentive

- 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.

State solution



State participants

- **Context**
  - defines the interface of interest to clients.
  - maintains an instance of a ConcreteState subclass that defines the current state

- **State –** defines a interface for encapsulating the behavior associated with a particular state of the Context.

- **ConcreteState subclasses**
  – each subclass implements a behavior associated with a state of the Context.

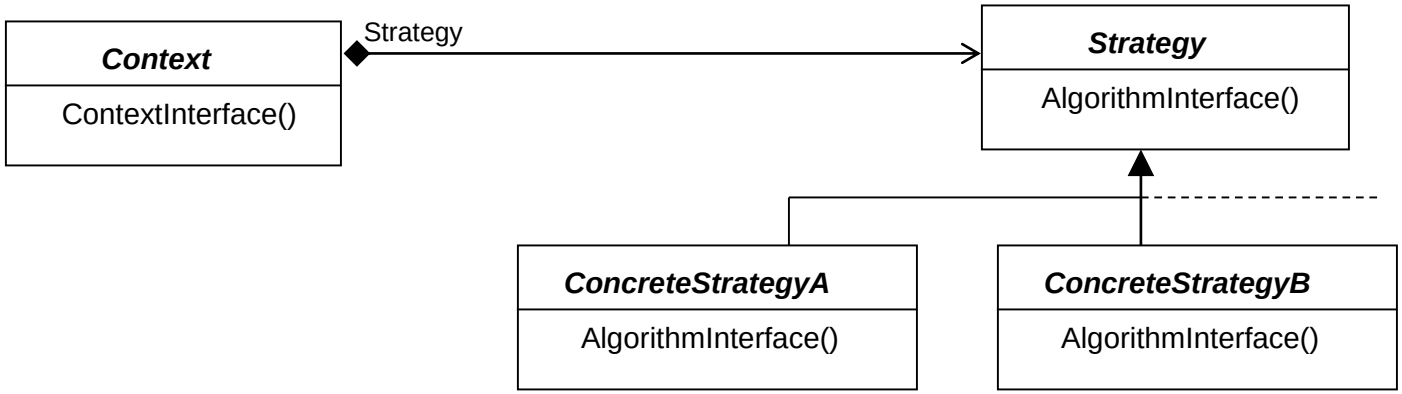
Strategy

- **“separate the invariant and variant behaviors of a system".**

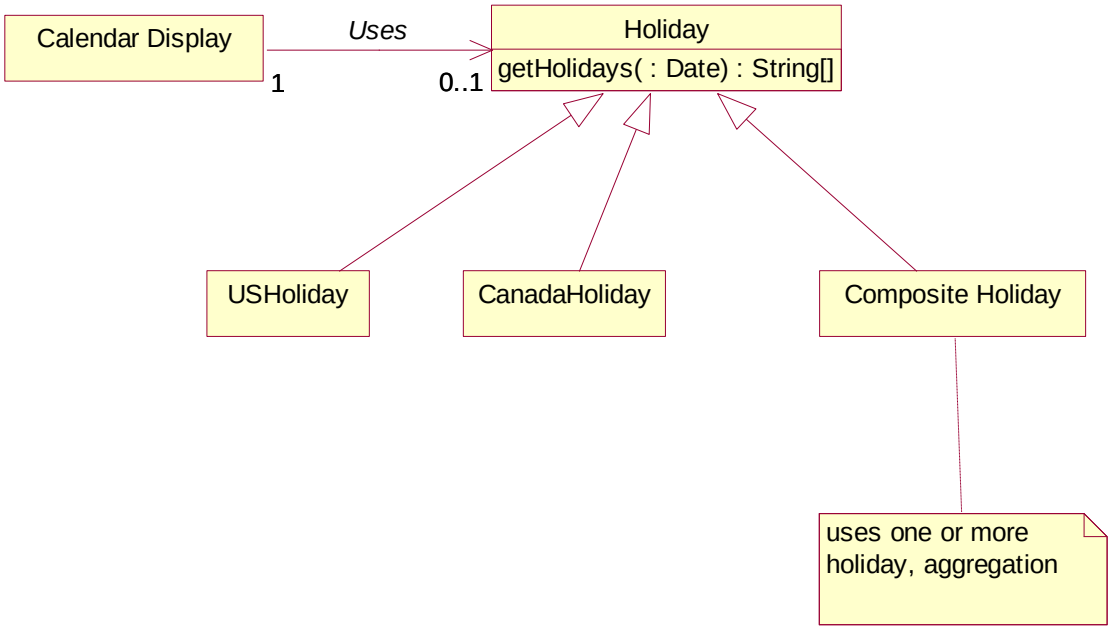
Incentive

- Use Strategy when:
  - many related classes differ only in their behavior.
  - you need different variants of an algorithm.
  - an algorithm uses data that clients shouldn’t know about.

Strategy solution



Strategy Example



Participants

- **Strategy**
  – declares an interface common to all supported algorithms.

- **ConcreteStrategy**
  – implements the algorithm using the Strategy interface.
- **Context**

  – is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

Command

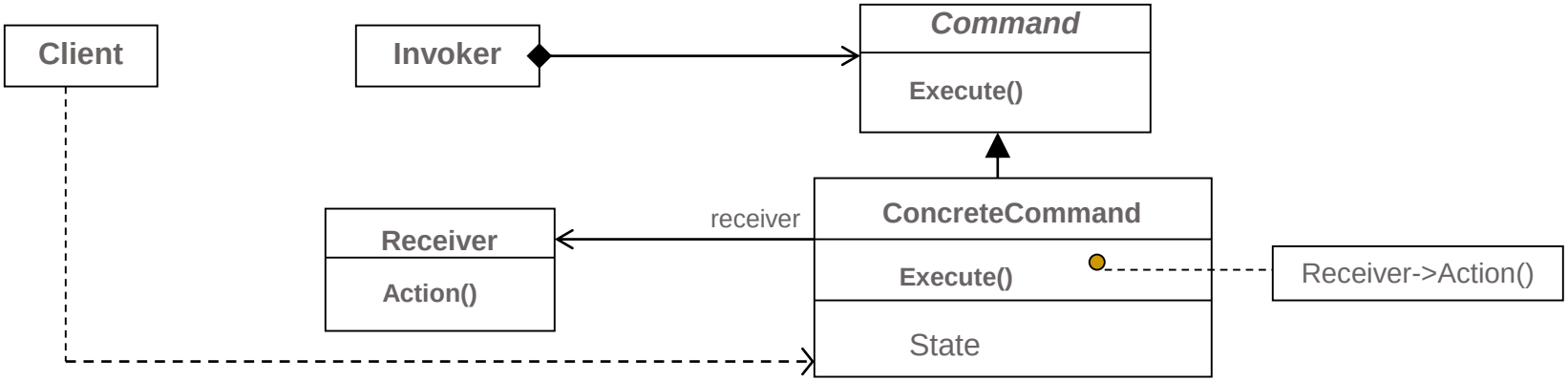
- **Intent:**

  Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Incentive

- To parameterize objects by an action to perform.
- Specify, queue, and execute requests at different times.
- Support undo.
- Support logging changes so that they can be reapplied in case of a system crash

Command solution



Participants

- **Command** – declares an interface for executing an operation.
- **ConcreteCommand** – defines a binding between a Receiver object and an action.
- **Client** – creates a ConcreteCommand object and sets its receiver.
- **Invoker**
  – asks the command to carry out the request.
- **Receiver** – knows how to perform the operations associated with carrying out a request. Any class may server a Receiver.

Usage in Java

- **MVC model 2**
  - Servlet act as a controller
  - Command pattern is used for
    - clear delegation to the model
    - View dispatching

Mediator

Intent:

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

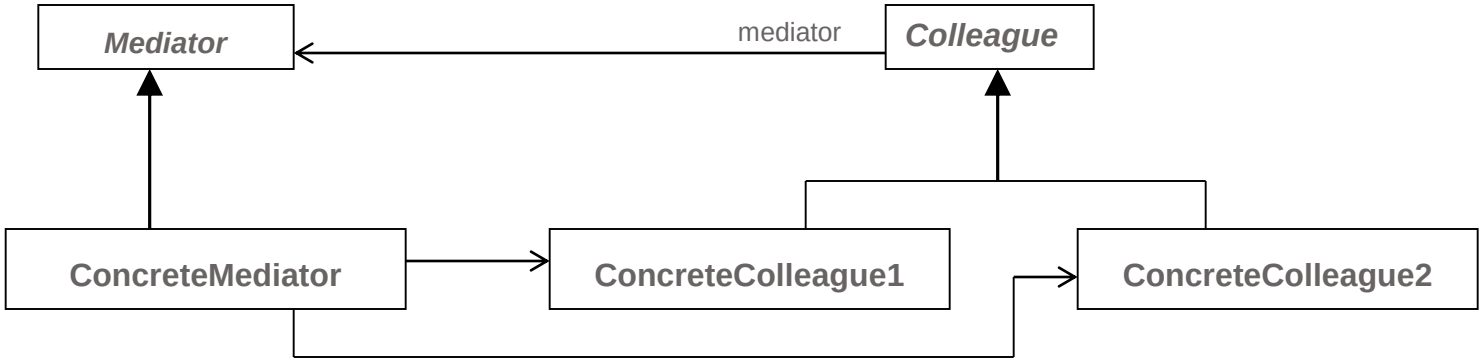
Mediator

“Objects interacting through a central controller, instead of interacting with each other directly".

Incentive

- A set of objects communicate in well-defined but complex ways.
- Reusing an objects is difficult because it refers to and communicates with many other objects.
- A behavior that’s distributed between several classes should be customizable without a lot of subclassing.

Mediator


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Participants

- **Mediator –** defines an interface for communicating with Colleague objects.

- **ConcreteMediator**
  – implements cooperative behavior by coordinating Colleague objects..
  - knows and maintains its colleagues.

- **Colleague classes**
  – each Colleague class knows its Mediator object.
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

Observer Pattern

- Creates a publish-subscribe relationship.
- Observers can register to receive events from the subject

Goal

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

The Problem

- In the system, a few modules from different application, need to be warned upon system memory get too low.

Solution 1

- Each module, sporadically , polls the system using system API.
- Problems:
  - Modules use complicated algorithm- doing its job and polling,
  - Each module must know the system API (portability)
  - The system is polled by a few modules

The Observer Solution

- The subject is the only element to poll the memory
- All observers attach/detach themselves to the subject
- Upon event, the subjects iterates through the attached observers.

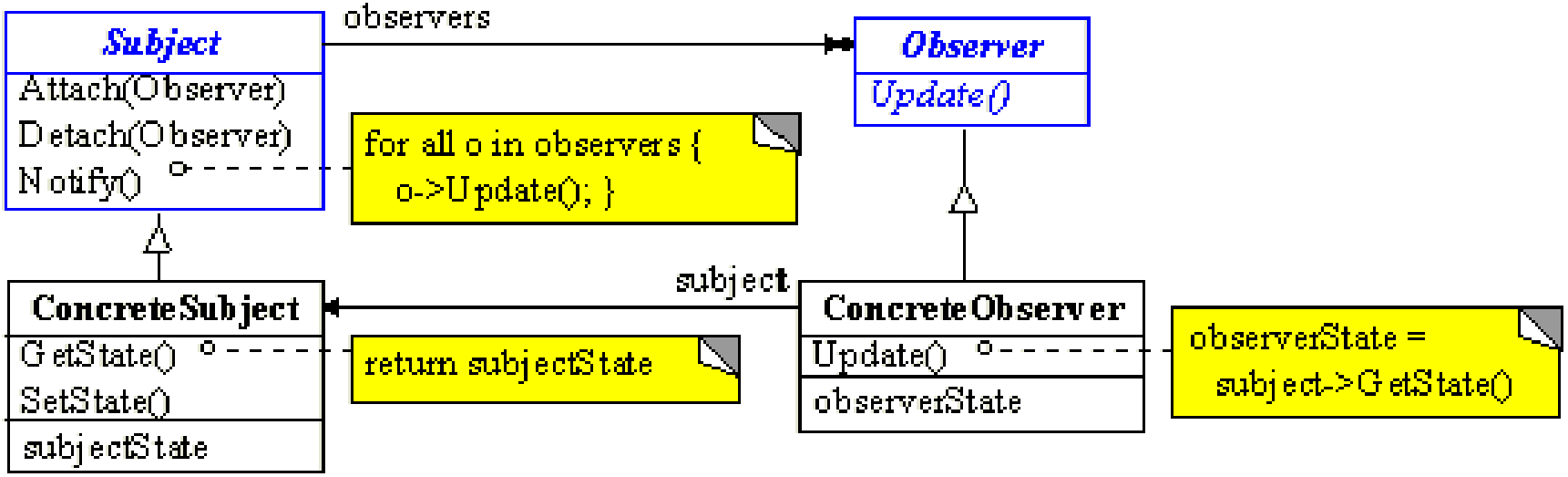
In General Terms

- The catch here is ‘a one-to-many dependency relationship between objects’.
- It decouples the observer from the subject.
- The subject knows nothing about its observers.

Incentive

- The subject informs the observers of a change
- The observers may sync themselves up with the latest state of the changed system.

The Observer Solution



Who triggers the update?

- Two options: either the subject notifies the observer or the client does the notify job.
- Pro- The client calling the ‘notify’ after a series of changes, which otherwise would have generates several updates by the subject.
- Con- The cost of client calling the ‘notify’ is to remember when to call it

Push and pull

- The notify() informs the observer about a change, not what the change is.
  - Here the observer is pulling the information from the subject.
- In the push changed data is sent as part of the notification.

Choosing specific changes to observe

- Where many of the changes in the subject will not affect the observers it would be efficient if the observer let the subject know what exactly the observer is interested in.

Code Example

Subject.java

```
public interface Subject
{
    public void register(Observer obsInst);
    public void unregister(Observer obsInst);
    public void notifyObservers(int data);
}
```

Code Example

```
public class ConcreteSubject implements Subject
{
        private Vector observersList = new Vector();
        public void register(Observer obsInst){
                boolean alreadyPresent = false;
                //check if it is already registered
                for ( int i = 0; i < observersList.size(); i++) {
                        if ((Observer)observersList.elementAt(i) == obsInst){
                                alreadyPresent = true;
                                break;
                        }
                }
                if (!alreadyPresent){
                    System.out.println("ConcreteSubject: Registering observer ...");
                        observersList.add(obsInst);
                }else
                   System.out.println("ConcreteSubject: Observer already registered ...");
        }
        public void notifyObservers(int data)
        {
                if (observersList.size() == 0)
                        System.out.println("ConcreteSubject: No observers to notify");
                else
                        System.out.println("ConcreteSubject: Notifying observers of change ... "+data);
                for ( int i = 0; i < observersList.size(); i++) {
                        ((Observer)observersList.elementAt(i)).update(data);
                }
        }
}
```

Code Example

Observer.java

```
public interface Observer
{
    public void update(int data);
}
```

PiechartConcreteObserver.java

```
import java.awt.*;
public class PiechartConcreteObserver extends Canvas implements Observer
{
    public void update(int data)
    {
        this.value = data;
        repaint();
    }
}
```

Code Example

client code

```
PiechartConcreteObserver pco = new PiechartConcreteObserver();
ConcreteSubject  concreteSubject = new ConcreteSubject();
ConcreteSubject.register(pco);
```

Visitor Pattern

- Provide logic in classes to support the operation.

Visitor Pattern

- An alternative way to implement operations that avoid complicating a structure
- Allows the logic to be varied by using different visitor classes.

Goal

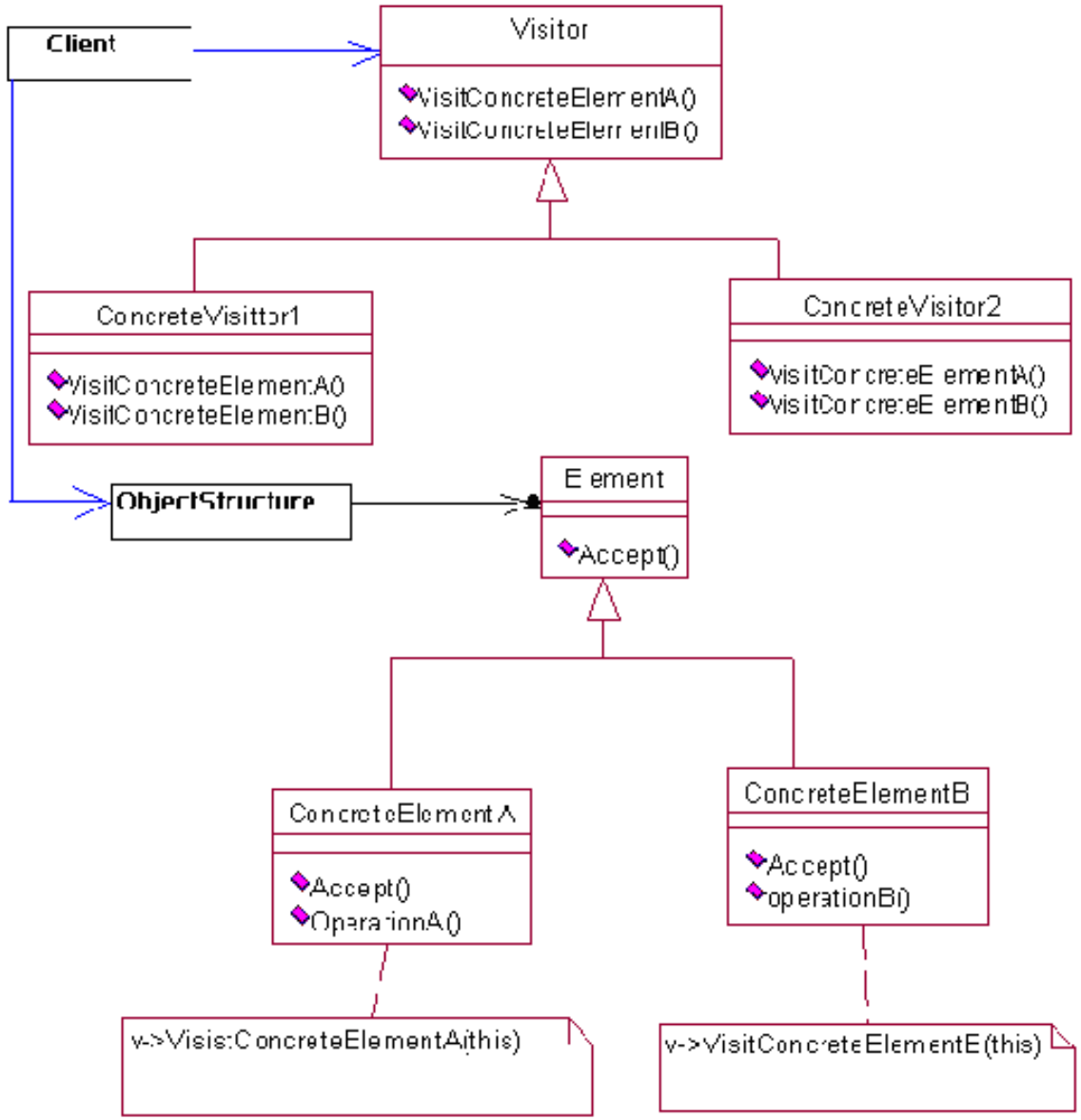
- Represent an operation to be performed on the elements of an object structure.
- Visitor lets you define a new operation without changing the classes of the elements on which it operates.

In General Terms

- How to "separate the structure of an object collection from the operations performed on that collection“?
- Adding new object types requires modifying the visitor that have already been written.

Example- Story

- A Store gives discounts on different *articles* depending on price range. These discounts would be varying depending on seasons. The manager wants to know what is current value of the articles in the store after applying the discounts.
- If we write discount function on *article*, we need to *keep changing when discounts changes*.


Client
Visitor
VisitConcreteElementA()
VisitConcreteElementB()
ConcreteVisitor1
VisitConcreteElementA()
VisitConcreteElementB()
ConcreteVisitor2
VisitConcreteElementA()
VisitConcreteElementB()
ObjectStructure
Element
Accept()
ConcreteElementA
Accept()
OperationA()
ConcreteElementB
Accept()
operationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Structure

- Visitor Design Pattern has these structure elements:
  - **Visitor** defines a Visitor operation for each ConcreteElement class.
  - **ConcreteVisitor** Implements visitor operations.
  - **Element** contains an Accept operation that would take a visitor as an argument.
  - **ConcreteElement** implements Accept operations.
  - **ObjectStructure** provides interface to Visitor.

Result?

- The following are the consequences of the Visitor Pattern:
  - It makes adding a new operation easy
  - It gathers related operations.
  - It make adding a new ConcreteElements hard
- It may force a break in encapsulation!

In Simpler Terms

- Double dispatch=Visitor pattern lets adding operations without changing classes.
- No static operations binding.
- Extending the Element interface by defining a new Visitor
- Who is responsible for traversing the object structure?

Related Patterns

- Iterator
- Composite

Code Example

```
public class Employee{
int sickDays, vacDays;
float Salary;
String Name;
public Employee(String name, float salary,int vacdays, int sickdays){
       vacDays = vacdays; sickDays = sickdays;
       Salary = salary; Name = name;
}
public String getName()
      { return Name; }
public int getSickdays()
      { return sickDays; }
public int getVacDays()
      { return vacDays; }
public float getSalary()
      { return Salary; }

public void accept(Visitor v)
      { v.visit(this); }
}
```

Code Example

```
/*
Note that we have included the accept method in this class. Now let's suppose that we want to prepare a report of the number
        of vacation days that all employees have taken so far this year. We could just write some code in the client to sum the
        results of calls to each Employee's getVacDays function, or we could put this function into a Visitor.
Since Java is a strongly typed language, your base Visitor class needs to have a suitable abstract visit method for each kind of
        class in your program. In this first simple example, we only have Employees, so our basic abstract Visitor class is just
*/
public abstract class Visitor
{
        public abstract void visit(Employee emp);
}
/*
Notice that there is no indication what the Visitor does with teachclass in either the client classes or the abstract Visitor class.
        We can in factwrite a whole lot of visitors that do different things to the classes in our program. The Visitor we are going to
        write first just sums the vacation data for all our employees:
*/
public class VacationVisitor extends Visitor
{
        protected int total_days;
        public VacationVisitor() { total_days = 0; }
        public void visit(Employee emp)
        {
                total_days += emp.getVacDays();
        }
                public int getTotalDays()
        {
                return total_days;
        }
}
```

Code Example

```
/*
Visiting the Classes
Now, all we have to do to compute the total vacation taken is to go
through a list of the employees and visit each of them, and then ask the
Visitor for the total.
*/
    VacationVisitor vac = new VacationVisitor();
    for (int i = 0; i < employees.length; i++)
    {
        employees[i].accept(vac);
    }
    System.out.println(vac.getTotalDays());
```

Exercise

Specify the correct DP to solve the problem described

1

- You want "platform-independent"
- Independent of how its parts are created, composed, and represented.
- Configured with one of multiple families of products.
- A family of products, designed to be used together, shall have their relationship constraints enforced.
- "case" statements are a maintenance nightmare - they shall be avoided.

?

2

- A subsystem shall not have the "type" of component it is responsible for creating hard-wired.
- The system shall be "open for extension, but closed for modification".

?

3

- A single instance shall be enforced, and the subsystem shall be responsible for it
- The instance shall be globally accessible.

?

4

- A legacy component shall be reused in a new design, and the interface "impedance mismatch" shall be reconciled.
- A reusable component shall cooperate with unrelated (or unforeseen) application components.

?

5

- The system supports recursive composition.
- The system supports "whole-part" hierarchical assemblies of components.
- Clients shall be able to transparently interact with compositions of components and individual components

?

6

- Components is extensible at run-time.
- The client shall be capable of configuring any combination of capabilities by simply specifying the layers (or wrappers, or onion skins) to be applied.

?

7

- The system provides a simple interface to a complex subsystem.
- The system provides alternative novice, intermediate, and "power-user" interfaces.
- The system decouples subsystems from each other.

?

8

- The system supports "distributed processing" by providing a local representative for a component in a different address space.
- The system supports "lazy creation" - a component is created only if, and when, the client demonstrates an interest in it.
- The system controls access to components by interposing an intermediary that evaluates the identity and access privileges of the requestor.
- The system architecture is characterized by multiple dimensions of indirection that offer a locus for intelligence
- The system allows the client to issue a request to one of several "handlers" without knowing or specifying the receiver explicitly.
- The system lets a suite of "handlers" to be configured at run-time.
- The system lets the client to "launch and leave" a request
- "Senders" and "receivers" are decoupled
- Chain of Responsibility

?

9

- The system architecture provides a "callback" framework.
- The system encapsulates "execute" requests to be created, queued, and serviced. They may also be logged, archived, loaded, and re-applied.
- The system shall support hierarchical compositions of primitive "command" abstractions.
- The system supports transactions
- "Senders" and "receivers" are decoupled

?

10

- The system supports accessing an aggregate component's contents without exposing its internal representation.
- The system supports multiple simultaneous traversals of aggregate components without complicating the implementation of the aggregate itself.
- The system defines a uniform interface for traversing dissimilar aggregate components.
- The system shall decouple "data structures" from "algorithms" so that each can be developed, maintained, and used independent of the other.

?

11

- The system encapsulates complex, many-to-many coupling between components in a separate component capable of allowing the "peers" to be: disengaged, replaced, and reused.
- The system supports numerous many-to-many "mappings" to be exchanged by the client.
- The system balances the distribution of intelligence emphasized by "logical" OO design with the centralization of intelligence often required by "physical" large scale design.
- "Senders" and "receivers" are decoupled

?

12

- The system supports multiple "views" of the same "model"
- Each "model" component is decoupled from the number and type of its "view" components.
- Each "view" component is capable of driving the flow of information from the "model" to itself.
- "Independent" components is decoupled from "dependent" components.
- "Senders" and "receivers" are decoupled from one another.

?

13

- Components are capable of "morphing" their behavior at run-time.
- The system architecture is characterized by a "finite state machine". The state machine needs to support application logic at each state transition, not simply the transition itself.
- "case" statements are a maintenance nightmare - they shall be avoided.

?

14

- Clients are decoupled from "choice of algorithm".
- Clients are decoupled from complex, algorithm-specific data structures.
- The choice, or implementation, of algorithm shall be configurable at run-time.
- "case" statements are a maintenance nightmare - they shall be avoided.

?

Answers

1. Abstract factory
2. Factory method
3. Singleton
4. Adapter
5. Composite
6. Decorator
7. Façade
8. Proxy
9. Command
10. Iterator
11. Mediator
12. Observer
13. State
14. Strategy