Object Oriented Analysis and Design using Java

UE21CS352

Prof. Vinay Joshi
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352: Object Oriented Analysis and Design using Java

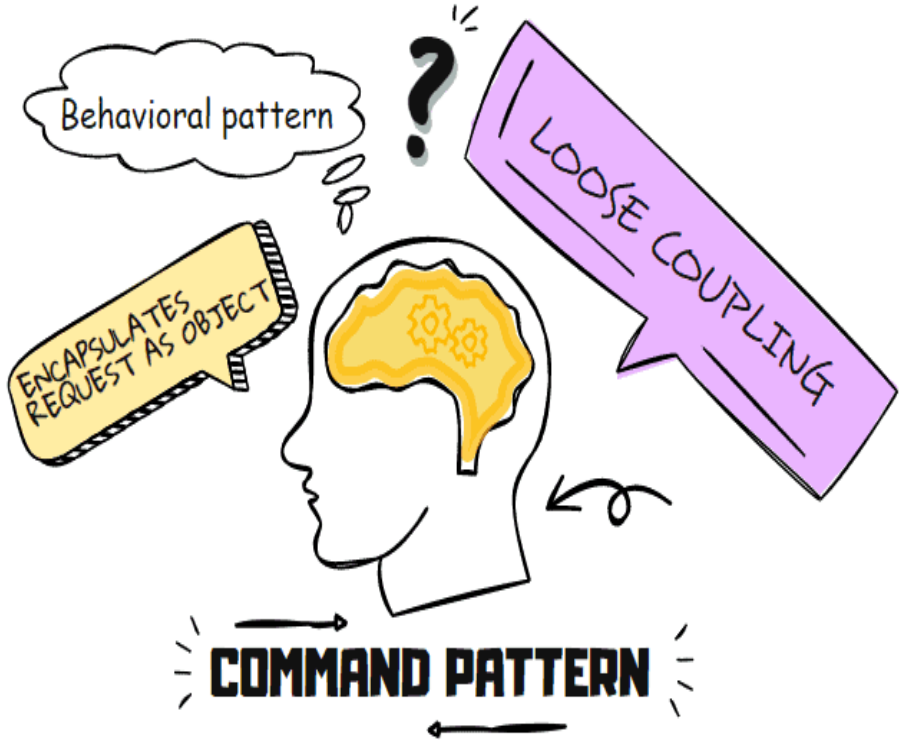
OO Behavioral Design Patterns with Sample implementation in Java

Prof. Vinay Joshi

Department of Computer Science and Engineering

Unit-4

Behavioral Patterns – Command Pattern



Command Pattern

- Command pattern is a data-driven design pattern and falls under behavioral patterns.
- A request is wrapped under an object as command and passed to an invoker object.
- The Invoker then searches for the appropriate object to handle the Command and passes it to the corresponding object that can execute it.
- The object, called Command, contains all the information needed to perform an action or trigger an event, such as the method name, the method owner, and the arguments for method parameters.

Intent

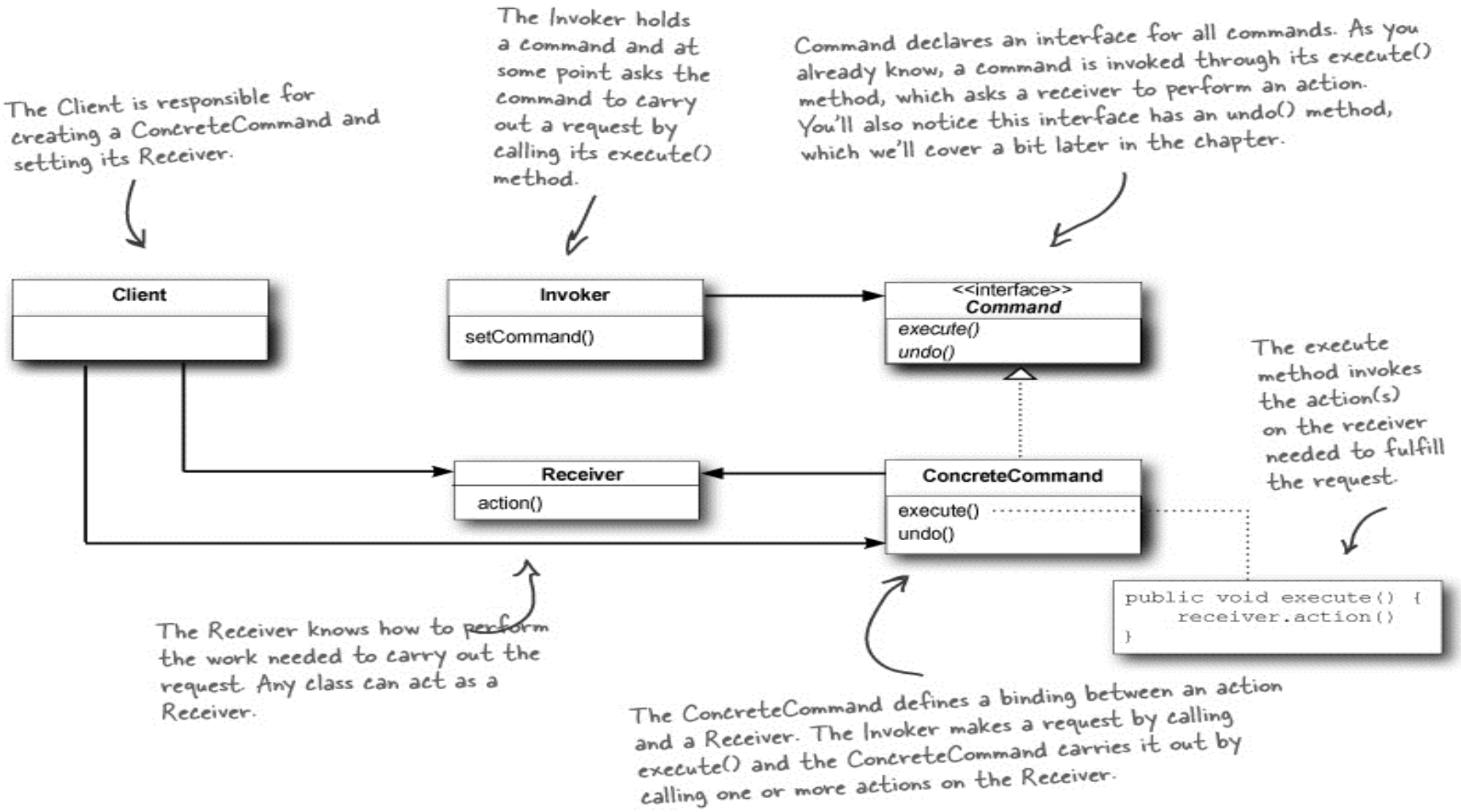
- encapsulate a request in an object
- allows the parameterization of clients with different requests
- allows saving the requests in a queue

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.

Object Oriented Analysis and Design using Java

Structure



Participants

Command: Declares an interface for executing an operation.

ConcreteCommand

- Defines a binding between a Receiver object and an action.
- Implements execute() by invoking a corresponding operation on Receiver.

Client (Application): Creates a Command object and sets its Receiver.

Invoker: Asks the Command to carry out a request.

Receiver: Knows how to perform the operation associated with a request. Can be any class.

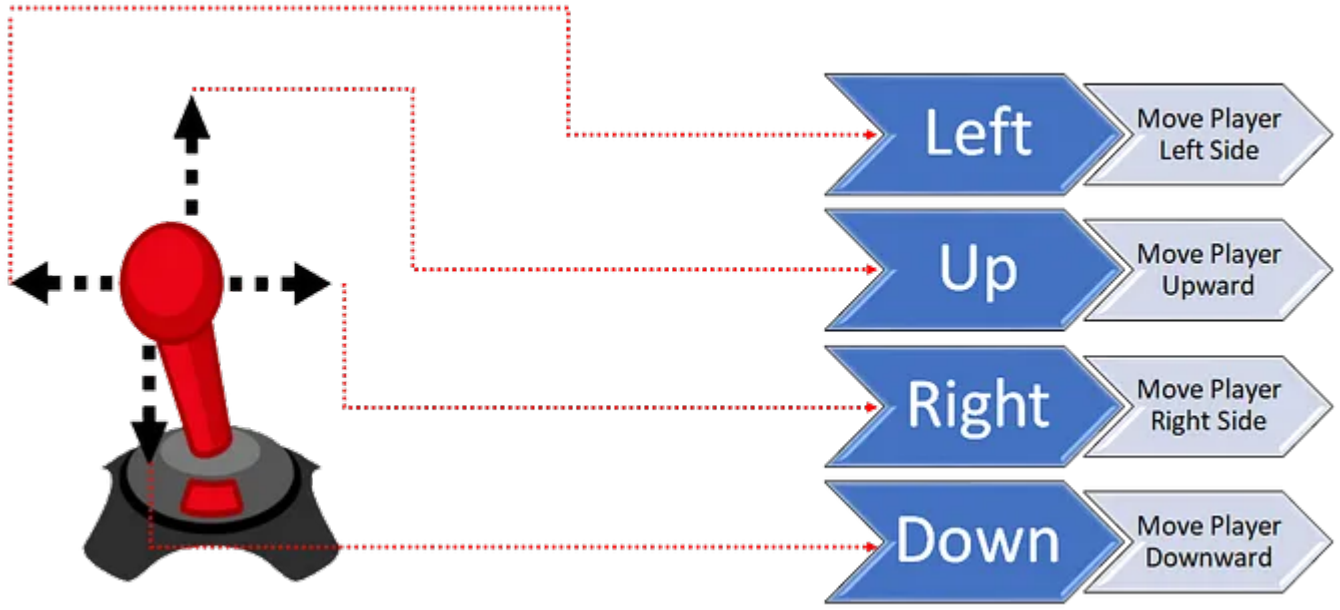
Collaborations

The Client creates a ConcreteCommand object and specifies its receiver.
An Invoker object stores the ConcreteCommand object.
The invoker issues a request by calling execute( ) on the command. When commands are undoable, ConcreteCommand stores state for undoing prior to invoking execute( ).
The ConcreteCommand object invokes operations on its receiver to carry out the request.

Example – Scenario 1

Command pattern helps to control the movement of the video game player using a joystick.

Based on the inputs being received for joysticks, we can invoke the commands which will move the player.



Consequences

Command decouples the object that invokes the operation from the one that knows how to perform it.

Commands are first-class objects. They can be manipulated and extended like any other object.

Its easy to add new Commands, because you don't have to change existing classes.

Commands can be assembled into a composite command.

Command Pattern has the following Limitations

- Must create a concrete command for each request
- May end up with large number of objects if the classes were not managed carefully

Applicability

parameterize objects by actions they perform(a way to implement callbacks)

specify, queue, and execute requests at different times (Partitioning complex requests)

Application that needs to support an undo command (single or multi-level)

Need to support change log for recovery purposes

Transaction based applications (Extendable)

Applications that learn about the receiver dynamically

Implementation – Example Scenario 2

Let us think of placing orders for buying and selling stocks. To implement the solution create an interface Order which acts as a command. Next create Stock class which acts as a request.

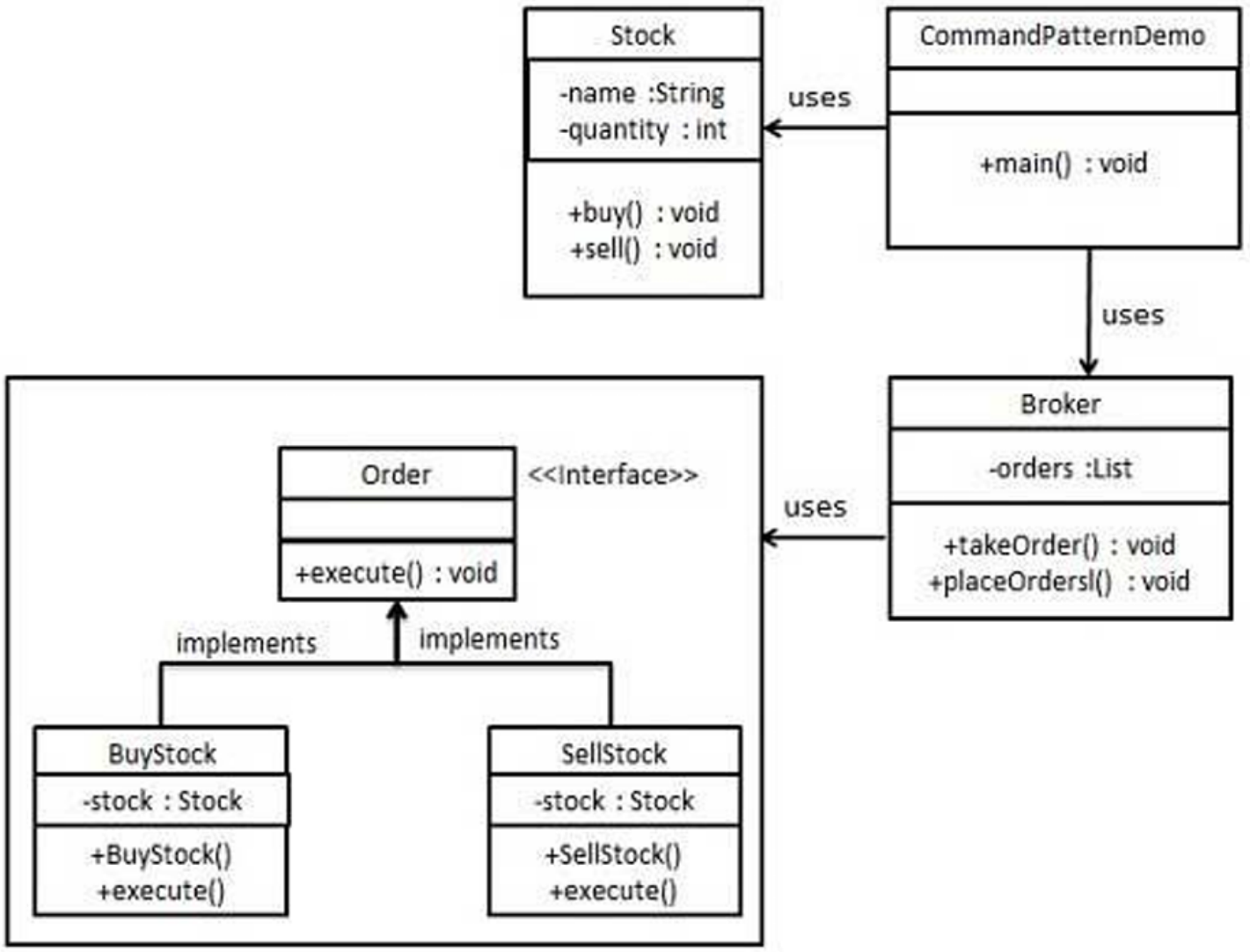
Then create the concrete command classes BuyStock and SellStock by implementing Order interface. These two classes do the actual command processing.

Create a class Broker which acts as an invoker object. It can take and place orders. Broker object uses command pattern to identify which object will execute which command based on the type of command.

Finally create a client CommandPatternDemo class for placing orders to buy and sell the stocks to demonstrate command pattern using broker.

Object Oriented Analysis and Design using Java

UML Class Diagram



Step 1 Create a command interface.
Order.java

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

Step 2 Create a request class or Receiver
Stock.java

```
public class Stock
{
   private String name = "ABC";
   private int quantity = 10;
   public void buy() {
      System.out.println("Stock [ Name: "+name+",  Quantity: " + quantity +" ] bought");
   }
   public void sell() {
      System.out.println("Stock [ Name: "+name+",  Quantity: " + quantity +" ] sold");
   }
}
```

Step 3 Create concrete classes implementing the Order interface.

BuyStock.java

```
public class BuyStock implements Order
{
   private Stock abcStock;
   public BuyStock(Stock abcStock) {
      this.abcStock = abcStock;
   }
   public void execute() {
      abcStock.buy();
   }
}
```

SellStock.java

```
public class SellStock implements Order
{
    private Stock abcStock;
    public SellStock(Stock abcStock){
        this.abcStock = abcStock;
    }
    public void execute() {
        abcStock.sell();
    }
}
```

Step 4 Create command invoker class.

Broker.java

```
import java.util.ArrayList;
import java.util.List;
public class Broker
{
    private List<Order> orderList = new
ArrayList<Order>();
    public void takeOrder(Order order){
       orderList.add(order);
    }
    public void placeOrders(){
       for (Order order : orderList){
          order.execute();
       }
       orderList.clear();
    }
}
```

Step 5 Client class. Use the Broker class to take and execute commands.
CommandPatternDemo.java

```
public class CommandPatternDemo
{
    public static void main(String[] args) {
        Stock abcStock = new Stock();
        BuyStock buyStockOrder = new BuyStock(abcStock);
        SellStock sellStockOrder = new SellStock(abcStock);

        Broker broker = new Broker();
        broker.takeOrder(buyStockOrder);
        broker.takeOrder(sellStockOrder);
        broker.placeOrders();
    }
}
```

Output

```
Stock [ Name: ABC,  Quantity: 10 ] bought
Stock [ Name: ABC,  Quantity: 10 ] sold
```

Benefits and Drawbacks

Benefits:

- It decouples the classes that invoke the operation from the object that knows how to execute the operation
- It allows you to create a sequence of commands by providing a queue system
- Extensions to add a new command is easy and can be done without changing the existing code
- You can also define a rollback system with the Command pattern, for example, in the Wizard example, we could write a rollback method

Drawbacks:

- There are a high number of classes and objects working together to achieve a goal. Application developers need to be careful developing these classes correctly.
- Every individual command is a ConcreteCommand class that increases the volume of classes for implementation and maintenance.

References

Text Reference

Design Patterns: Elements of Reusable Object-Oriented Software, GOF

Web Reference

https://dzone.com/articles/design-patterns-command
https://refactoring.guru/design-patterns

THANK YOU

Prof. Vinay Joshi

Department of Computer Science and Engineering

vinayj@pes.edu

Object Oriented Analysis and Design using Java

UE21CS352

Prof. Vinay Joshi
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

PES UNIVERSITY
CELEBRATING 50 YEARS

UE21CS352: Object Oriented Analysis and Design using Java

Unit-4

Behavioural Patterns – Interpreter

Agenda

- What is an Interpreter design pattern?
- Intent and Motivation
- Applicability
- Design Solution
  - Structure
  - Example Implementation
- Advantages and Disadvantages

Interpreter design pattern

The Interpreter pattern is a **behavioral design pattern** that defines a way to interpret a domain-specific language or grammar. It allows you to define the grammar of a language and provide a way to evaluate sentences in that language.

Intend

1. Provide a way to interpret and execute expressions in that language.
2. Break down complex expressions into smaller components that can be easily interpreted.
3. Create parsers and compilers for the defined language.
4. Build scripting languages and other domain-specific languages.
5. Provide a flexible and extensible way to define and interpret expressions in the language.
6. Support the implementation of rules that govern the behavior of the language.
7. Facilitate the creation of abstract syntax trees to represent expressions in the language.
8. Allow for the creation of interpreters that can handle different types of expressions and operators.

Motivation

1. **Define a domain-specific language:** It allows developers to define a language that is specific to a particular domain or problem. This language can be tailored to meet the needs of a specific application, making it easier to write code that is expressive and concise.
2. **Interpret and execute expressions dynamically:** It provides a way to interpret and execute expressions at runtime. This is useful when dealing with complex expressions that are difficult or impossible to evaluate statically.
3. **Separate concerns:** It separates the concerns of defining a language and interpreting expressions in that language. This makes it easier to modify the language and add new expressions without affecting the interpreter code.

Motivation

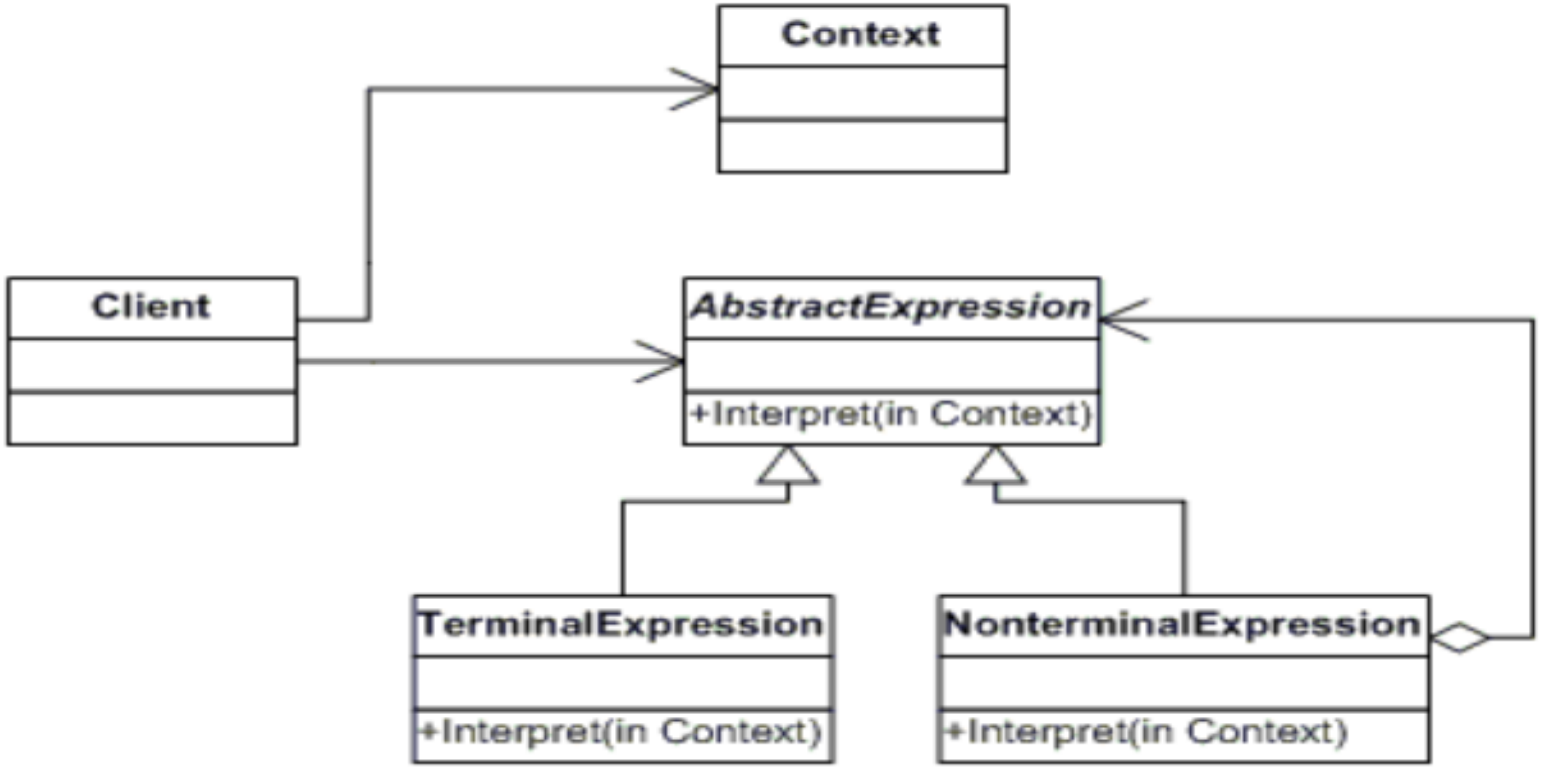
4. **Build parsers and compilers:**

The Interpreter pattern can be used to build parsers and compilers for the defined language. This allows the language to be used in a wide range of applications and contexts.

5. Facilitate communication between different components:

The Interpreter pattern can be used to facilitate communication between different components in a system. By defining a common language, different components can exchange messages and work together more easily.

Design Solution – Structure



Design Solution –Participants

AbstractExpression (Expression)

It declares an interface for executing an operation

TerminalExpression(ThousandExpression,HundredExpression,TenExpression, OneExpression)

It implements an Interpret operation associated with terminal symbols in the grammar.

An instance is required for every terminal symbol in the sentence.

NonterminalExpression (not used)

One such class is required for every rule R ::= R1R2...Rn in the grammar maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn. Implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.

Design Solution –Participants

Context (Context)

It contains information that is global to the interpreter

Client (InterpreterApp)

It builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes

It invokes the Interpret operation

Example -Implementation

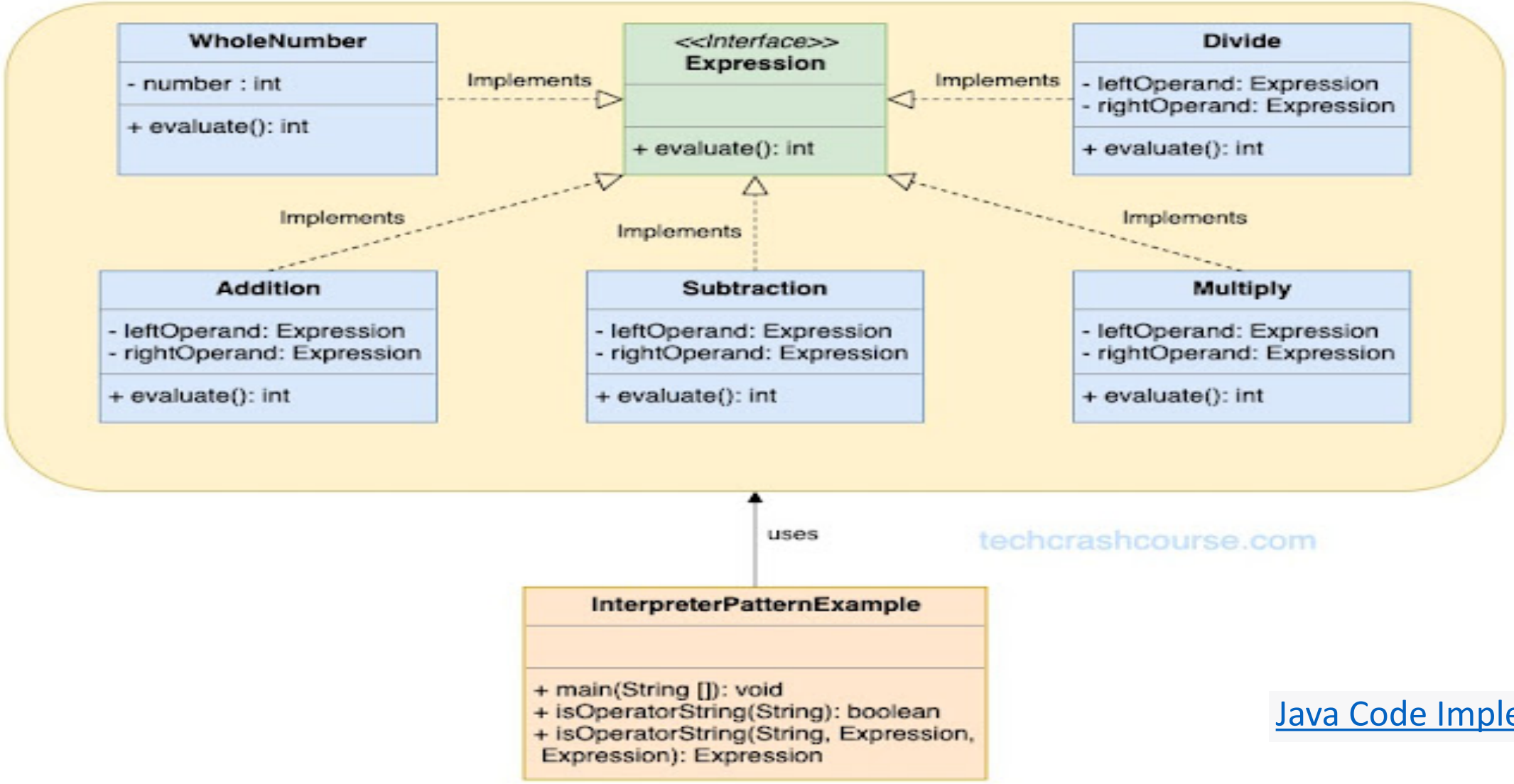
Let's take an example of an algebraic expression 2 + 3. To model this expression we need ConstantExpression that represents the constant operands(2 and 3) and then AddExpression that takes two ConstantExpression expressions operands and performs addition operation. All Expression classes must implement a common interface 'Expression'.

Above expression can Modelled as :
AddExpression(ConstantExpression(2), ConstantExpression(3))

Similarly, (2 + 6) + (1 + 9) can be modelled as:

```
AddExpression(
        AddExpression(ConstantExpression(2), ConstantExpression(6)),
AddExpression(ConstantExpression(1), ConstantExpression(9))
)
```

Example -Implementation



Java Code Implementation

Applicability

- The Interpreter pattern is often used in situations where you need to interpret a domain-specific language or grammar, such as in a compiler or an interpreter for a scripting language.

- It can also be used to implement rule-based systems, where rules are expressed in a domain-specific language.

Applications

- **Regular Expressions:** Regular expressions are a powerful tool for matching patterns in strings. An interpreter can be used to parse and execute regular expressions, making it easier to build complex matching patterns.
- **SQL Queries:** SQL queries can be complex and difficult to write. An interpreter can be used to parse and execute SQL queries, allowing developers to build complex queries without having to understand the underlying implementation details.
- **Mathematical Expressions:** Mathematical expressions can be difficult to parse and evaluate. An interpreter can be used to parse and evaluate mathematical expressions, allowing you to build complex calculations and formulas.
- **Programming Languages:** Interpreters are commonly used in programming languages to execute code. They can be used to interpret scripts, run-time code, or compile code on-the-fly.

Advantages

- **Flexibility:** The Interpreter pattern provides a flexible way to implement complex grammars or expressions. It allows you to define your own language and parse sentences in that language.
- **Extensibility:** The Interpreter pattern makes it easy to add new functionality or behavior to your application. You can simply add new expressions or modify existing ones to change the behavior of your program.
- **Separation of Concerns:** The Interpreter pattern separates the parsing and execution logic, making it easier to maintain and extend your code.

Disadvantages

- **Complexity:** The Interpreter pattern can add complexity to your code, especially when dealing with complex grammars or expressions.
- **Performance:** The Interpreter pattern can be slower than other approaches, such as compiling or using a parser generator, since it requires interpreting the expressions at run-time.
- **Maintenance:** The Interpreter pattern can make your code more difficult to maintain, especially if the language or expressions change frequently.

Consequence

- The Interpreter pattern is a powerful tool that can make your code more flexible and extensible.
- However, it also comes with some trade-offs in terms of complexity, performance, and maintenance. It's important to consider these factors when deciding whether to use the Interpreter pattern in your application.

References

- "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
- "Head First Design Patterns: A Brain-Friendly Guide" by Eric Freeman and Elisabeth Robson.
- "Design Patterns Explained: A New Perspective on Object-Oriented Design" by Alan Shalloway and James R. Trott.

THANK YOU

Prof. Vinay Joshi

Department of Computer Science and Engineering

vinayj@pes.edu

Object Oriented Analysis and Design using Java

UE21CS352

Prof. Vinay Joshi
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

PES UNIVERSITY
CELEBRATING 50 YEARS

UE20CS352: Object Oriented Analysis and Design using Java

Unit-4

Behavioral Patterns – Iterator

Agenda

- What is an Iterator design pattern?
- Intent and Motivation
- Applicability
- Design Solution
  - Structure
  - Participants
  - Example Implementation
- Concequences
- Advantages and Disadvantages

Iterator design pattern

- The Iterator design pattern is a **behavioral pattern** that provides a way to access the elements of an aggregate object sequentially, without exposing its underlying implementation.

- In other words, it provides a uniform way to access the elements of a collection, such as a list, set, or map, without having to know how the collection is implemented.

Intend

- The intent of using an iterator is to provide a standardized way of accessing and manipulating collections of data in a simple, efficient, and flexible manner.
- It is useful for implementing algorithms that require sequential access to data, such as sorting, searching, and filtering. By using an iterator, you can easily apply these algorithms to any collection of data, regardless of its specific implementation or data structure

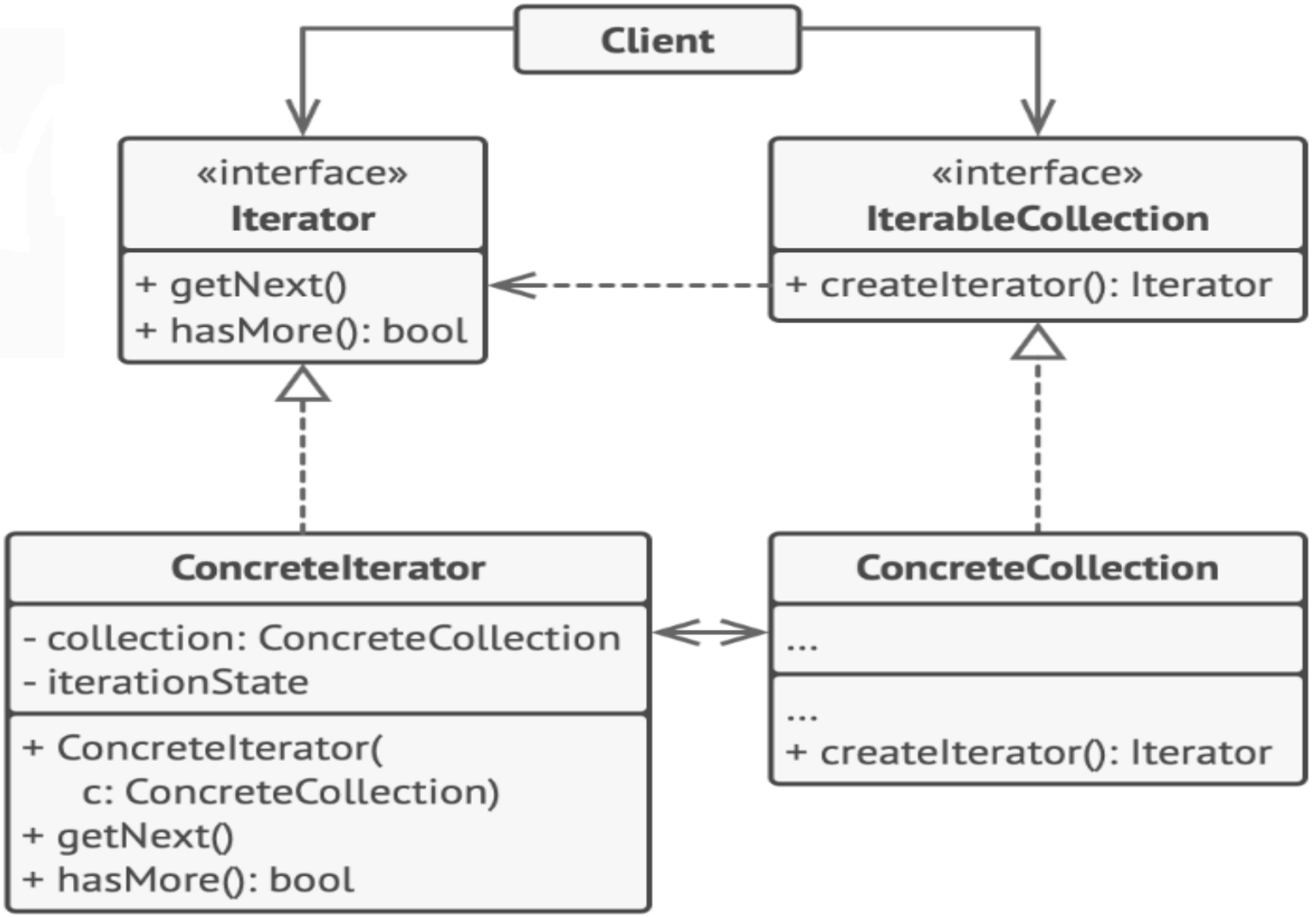
Motivation

- Encapsulation
- Abstraction
- Separation of concerns
- Lazy evaluation
- Multiple traversal support

Lazy evaluation

The Iterator pattern supports lazy evaluation of collections, where the elements of the collection are computed or retrieved only when they are needed. This can be especially useful for working with large datasets or data streams.

Design Solution – Structure



Design Solution –Participants

- **Iterator (AbstractIterator)**
  Defines an interface for accessing and traversing elements.
- **ConcreteIterator (Iterator)**
  Implements the Iterator interface.
  Keeps track of the current position in the traversal of the aggregate.
- **Aggregate (AbstractCollection)**
  Defines an interface for creating an Iterator object
- **ConcreteAggregate (Collection)**
  Implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Consequence

Consequences of using the Iterator design pattern are:

1. **Improved code flexibility:** The Iterator pattern allows clients to iterate over collections in different ways without changing the underlying collection's structure. This improves code flexibility and makes it easier to add new iteration methods in the future.
2. **Better encapsulation:** By providing a separate object to handle iteration, the internal details of the collection are hidden from the client code. This improves encapsulation and helps to prevent code from becoming tightly coupled.
3. **Simplified client code:** The Iterator pattern simplifies the client code by providing a standardized way to access collection elements. This reduces the complexity of the client code and makes it easier to read and maintain.
4. **Performance overhead:** The Iterator pattern may introduce a performance overhead, as the iteration process involves additional method calls and object creations. However, this overhead is typically negligible for small collections.

Overall, the Iterator design pattern is a useful tool for improving the flexibility, encapsulation, and maintainability of code that involves iterating over collections.

Applicability

- Accessing elements in a collection
- Simplifying complex data structures
- Implementing lazy evaluation
- Supporting multiple traversals
- Providing a standard interface

Accessing elements in a collection

The Iterator pattern is used to provide a simple and efficient way to access and iterate over the elements in a collection, such as an array or a list.

Advantages

- *Single Responsibility Principle*. You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes.

- *Open/Closed Principle*. You can implement new types of collections and iterators and pass them to existing code without breaking anything.

- You can iterate over the same collection in parallel because each iterator object contains its own iteration state.

- For the same reason, you can delay an iteration and continue it when needed.

Disadvantages

- Applying the pattern can be an overkill if your app only works with simple collections.

- Using an iterator may be less efficient than going through elements of some specialized collections directly.

References

- "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
- "Head First Design Patterns: A Brain-Friendly Guide" by Eric Freeman and Elisabeth Robson.
- "Design Patterns Explained: A New Perspective on Object-Oriented Design" by Alan Shalloway and James R. Trott.

THANK YOU

Prof. Vinay Joshi

Department of Computer Science and Engineering

vinayj@pes.edu

Object Oriented Analysis and Design using Java

UE21CS352

Prof. Vinay Joshi
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352: Object Oriented Analysis and Design with Java

OO Behavioral Design Patterns with Sample implementation in Java

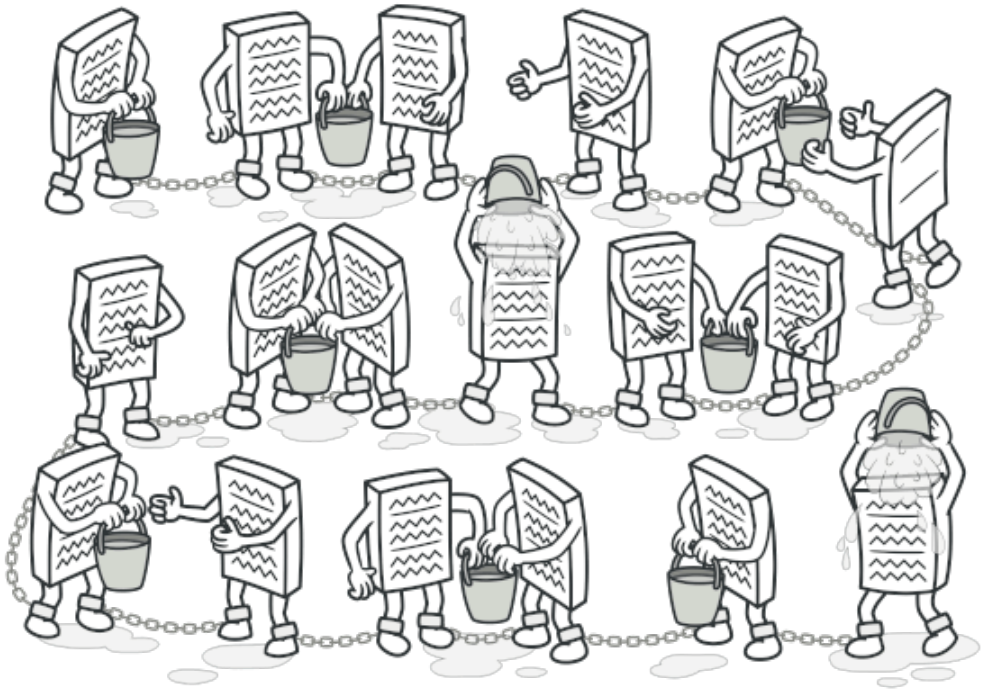
Prof. Vinay Joshi

Department of Computer Science and Engineering

PES UNIVERSITY
CELEBRATING 50 YEARS

Unit-4

Behavioral Patterns – Chain of Responsibility Pattern



Behavioral patterns

Behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns.

These patterns increase flexibility in carrying out this communication.

Behavioral patterns influence how state and behavior flow through a system.

By optimizing how state and behavior are transferred and modified, you can simplify, optimize, and increase the maintainability of an application

Object Oriented Analysis and Design using Java

List of common behavioral design patterns

1. **Chain of responsibility:** Command objects are handled or passed on to other objects by logic-containing processing objects
2. **Command:** Command objects encapsulate an action and its parameters
3. **Interpreter:** Implement a specialized computer language to rapidly solve a specific set of problems
4. **Iterator:** Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation

Mediator: Provides a unified interface to a set of interfaces in a subsystem

Memento: Provides the ability to restore an object to its previous state (rollback)

Observer: also known as Publish/Subscribe or Event Listener. Objects register to observe an event that may be raised by another object

State: A clean way for an object to partially change its type at runtime

Strategy: Algorithms can be selected on the fly

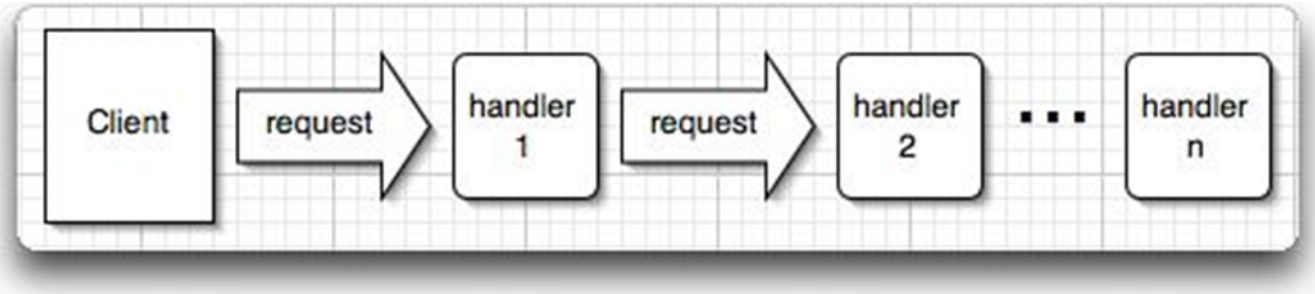
Template Method: Describes the program skeleton of a program

Visitor: A way to separate an algorithm from an object

Chain of Responsibility Pattern

The Chain of Responsibility pattern establishes a chain within a system, so that a message can either be handled at the level where it is first received, or be directed to an object that can handle it.

it is used to manage algorithms, relationships and responsibilities between objects



Intent

Avoid coupling sender-of-request to its receiver by giving more than one object a chance to handle request.

Chain receiving objects and pass request along until an object handles it.

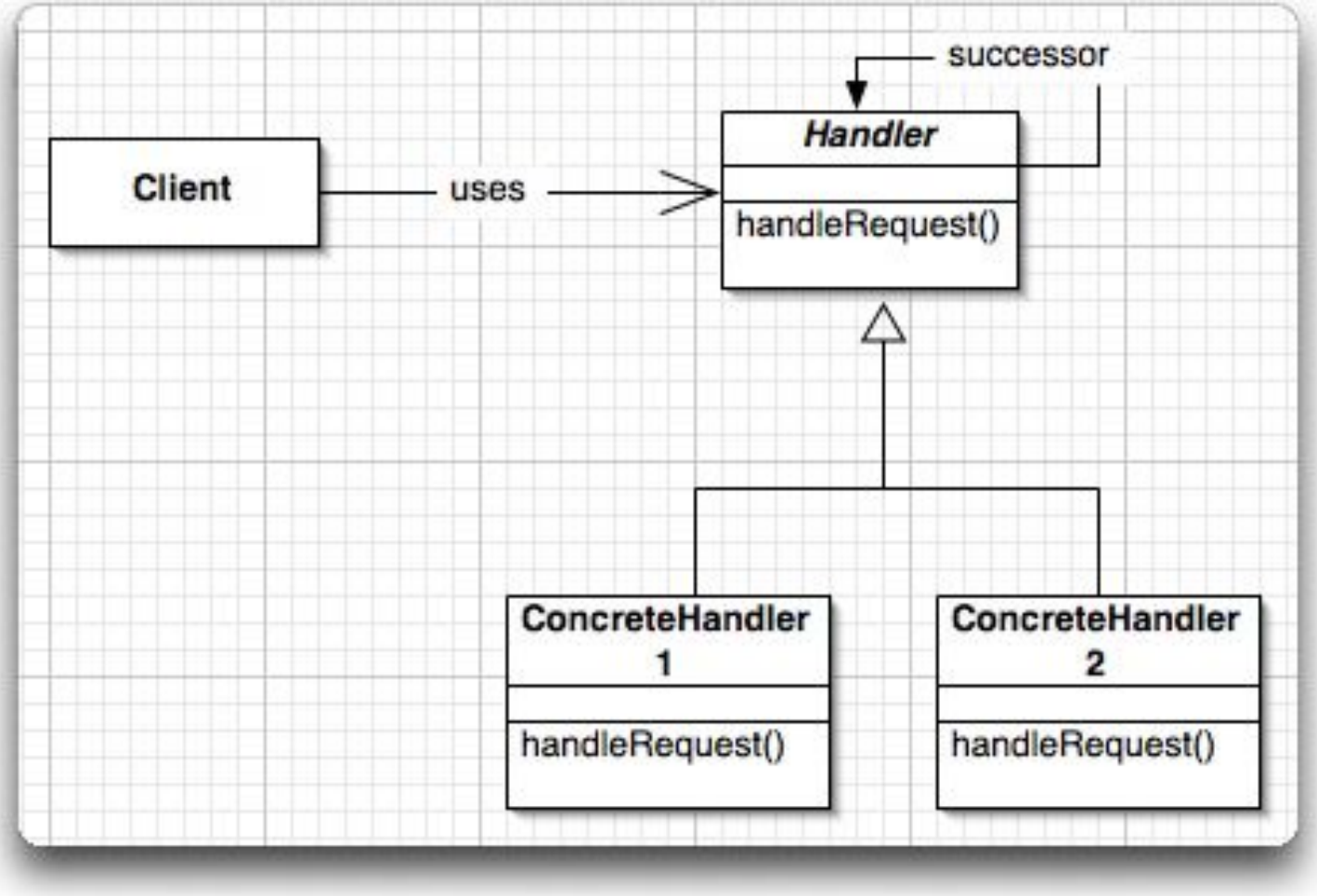
Motivation

The Chain of Responsibility is intended to promote loose coupling between the sender of a request and its receiver by giving more than one object an opportunity to handle the request.

The receiving objects are chained and pass the request along the chain until one of the objects handles it.

The set of potential request handler objects and the order in which these objects form the chain can be decided dynamically at runtime by the client depending on the current state of the application.

Structure


successor
Handler
handleRequest()
Client
uses
ConcreteHandler 1
handleRequest()
ConcreteHandler 2
handleRequest()

Participants

Handler:

- This can be an interface which will primarily receive the request and dispatches the request to a chain of handlers. It has reference to only the first handler in the chain and does not know anything about the rest of the handlers.

ConcreteHandler:

- handles requests it is responsible for
- can access its successor
- if it does not handle the request, forwards the request to its successor

Client:

- initiates the request to a ConcreteHandlerObject on the chain

Example – Scenario 1

Scenario

An enterprise has been getting more email than they can handle. The enterprise gets 4 types of e-mail.

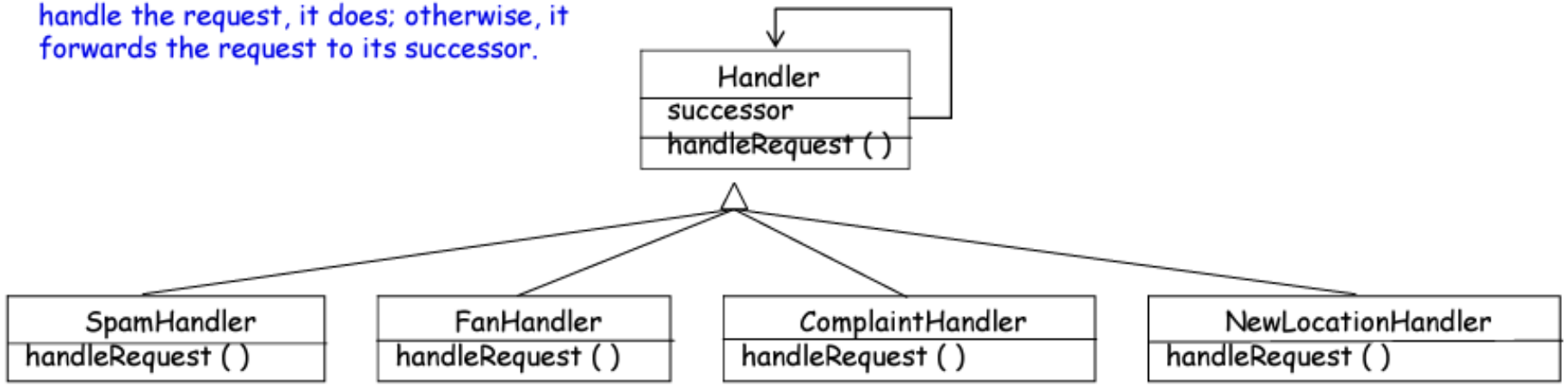
They are:

- Fan mail with compliments,
- Complaints,
- Requests for new features,
- Spam.

Your task:

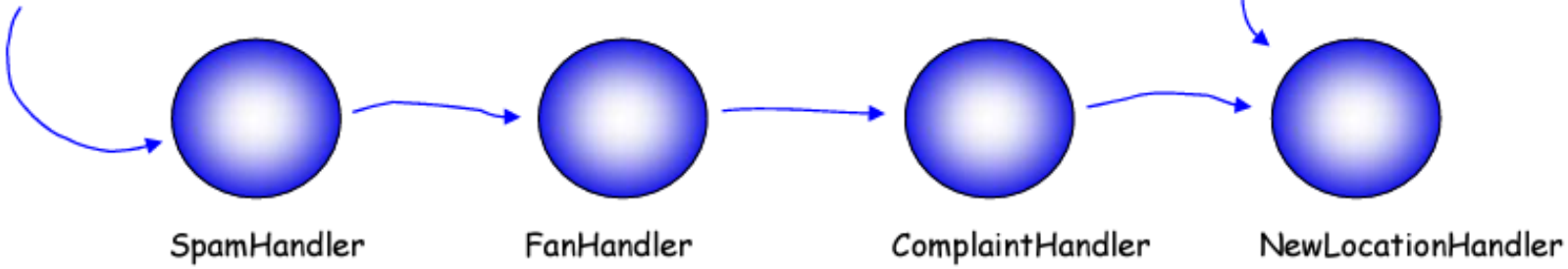
- The enterprise has already written the AI detectors that can tell whether an email is fan, complaint, request or spam,
- You need to create a design that can use the detectors to handle incoming email.

Each object in the chain acts as a handler and has a successor object. If it can handle the request, it does; otherwise, it forwards the request to its successor.



Each email is passed to the first handler

Email is not handled if it falls off the end of the chain -- although, you can always implement a catch-all handler



Consequences

Reduced Coupling

- Objects are free from knowing what object handles the request

Added Flexibility in assigning responsibilities to objects

- Can change chain at runtime
- Can subclass for special handlers

Receipt is guaranteed

- Request could fall off the chain
- Request could be dropped with bad chain

Applicability

Use Chain of Responsibility

when

- More than one object may handle a request and the handler isn't known a priori.
- You want to issue a request to one of several objects without specifying the receiver explicitly
- The Set of objects than can handle a request should be specified dynamically

Implementation – Example Scenario 2

Let us consider the transaction approval process in a company. Suppose we want to approve the transactions based on certain conditions?

For instance, in the transaction approval application user keys in transaction details into the application and this transaction need to be processed by any one of the higher level employee in the company.

All transactions lesser than 1,00,000 amount can be approved by manager, lesser than 10,00,000 can be approved by vice president and lesser than 25,00,000 can be approved by CEO.

The transaction approval behavior can be implemented with a simple if else conditions, by checking if amount is less than 1,00,000 or else if amount is less than 10,00,000 and so on. But this approach is static, means we can not change these conditions dynamically. Chain of responsibility design pattern can be used in this situation.

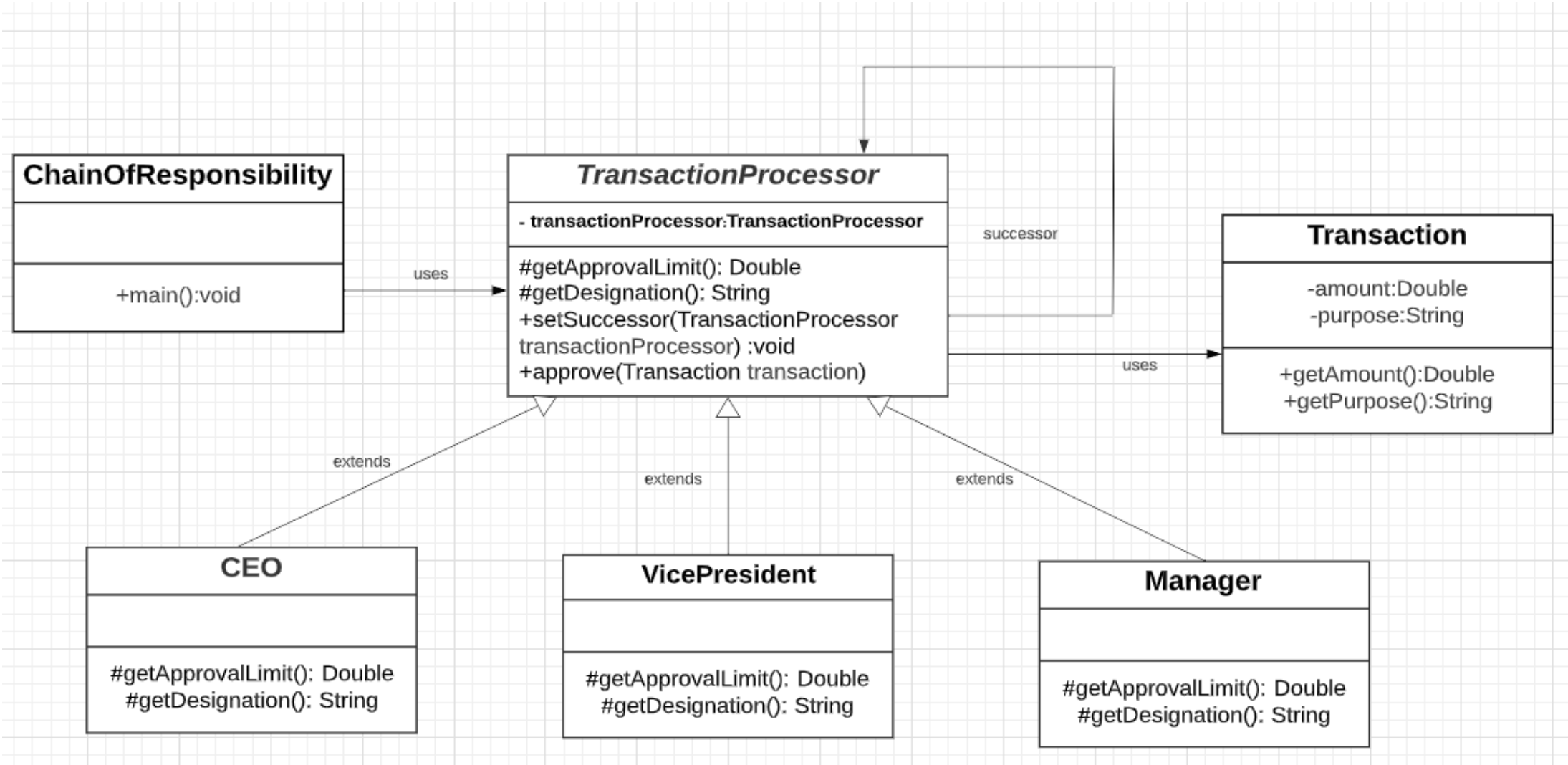
In the transaction processing application, approval process acts like a chain of responsibilities.

First, manager acts on the transaction, if the amount is higher than his approval limit then the transaction is transferred to vice president and so on.

We need to design processing and command objects. Based on the above example, processing objects are employees like Manager, Vice President because they process the transaction by approving and command object is the transaction. Once processing objects are created then we need to chain them together like manager -> vice president -> CEO and pass the transaction at the beginning of the chain. Transaction continues to flow in the chain until it reaches the employee, who's limit allows to approve it.

Object Oriented Analysis and Design using Java

UML Class Diagram



First create an abstract class for processing the transactions.

If the transaction can be processed then it should be passed to its successor so this class store

its next successor object.

If transaction is with in the limit then it prints the message.

If it is out of range then passes it to next processor by calling ‘approve’ method,

before calling approve check for null if processor request is last in the chain.

```
abstract class TransactionProcessor
{
    private TransactionProcessor transactionProcessor;
    abstract protected Double getApprovalLimit();
    abstract protected String getDesignation();
    public void setSuccessor(TransactionProcessor transactionProcessor)
    {
        this.transactionProcessor = transactionProcessor;
    }
    public void approve(Transaction transaction)
    {
       if(transaction.getAmount() <=0.0 )
        {
            System.out.println("Invalid Amount. Amount should be > 0");
            return;
        }
        if(transaction.getAmount() <= getApprovalLimit())
        {
            System.out.println("Transaction for amount "+transaction.getAmount()+ " approved by "+getDesignation());
        }
        else
        {
            if(transactionProcessor == null)
             {
                System.out.println("Invalid Amount. Amount should not exceed 25lacs!");
                 return;
               }
              transactionProcessor.approve(transaction);  } }}
```

Now we need to create concrete class for each approver (Manager, Vice President, CEO) and this class will extend TransactionProcessor.

Each processing class sets its limit and designation.

```
class Manager extends TransactionProcessor{

    @Override
    protected Double getApprovalLimit() {
        return 100000.0;
    }

    @Override
    protected String getDesignation() {
        return "manager";
    }
}
```

```
class VicePresident extends TransactionProcessor{

    @Override
    protected Double getApprovalLimit() {
        return 1000000.0;
    }

    @Override
    protected String getDesignation() {
        return “Vice President";
    }
}
```

```
class CEO extends TransactionProcessor{

    @Override
    protected Double getApprovalLimit() {
        return 2500000.0;
    }

    @Override
    protected String getDesignation() {
        return "CEO";
    }
}
```

Next Create a transaction class.

```
class Transaction{
    private Double amount;
    private String purpose;

    Transaction(Double amount, String purpose){
        this.amount = amount;
        this.purpose = purpose;
    }
    public Double getAmount() {
        return amount;
    }
    public String getPurpose() {
        return purpose;
    }
}
```

Now let us glue all these classes together to form chain of responsibility pattern using ChainOfResponsibility class.

- Create processing objects. We have three processing objects in the chain.
- Chain together all the processing objects. Order of the objects is important because manager handover transaction to vice president.
- Because the chain starts from the manager so transactions are pushed using the manager instance.

```
public class ChainOfResponsibility {
public static void main(String[] args) {
        Manager manager = new Manager();
        VicePresident vicePresident = new VicePresident
        CEO ceo = new CEO();
        manager.setSuccessor(vicePresident);
        vicePresident.setSuccessor(ceo);
        manager.approve(new Transaction(2600000.0, "general"
        manager.approve(new Transaction(50000.0, "general"))
        manager.approve(new Transaction(120000.0, "general"
        manager.approve(new Transaction(1500000.0, "general"
        manager.approve(new Transaction(0.0, "general"));
}
}
```

Creating Chain

manager class returns its approval limit amount. TransactionProcessor class approve method checks whether the given amount is <= approval limit. If it returns false it propagates the request to the successor VicePresident class to get its approval limit and so on. In case the request can not be serviced then null object will be return for the amount exceeding 25lacs.

Output

```
Invalid Amount. Amount should not exceed 25lacs!
Transaction for amount 50000.0 approved by manager
Transaction for amount 120000.0 approved by vice president
Transaction for amount 1500000.0 approved by CEO
Invalid Amount. Amount should be > 0
```

Benefits, Uses and Drawbacks

Benefits:

- Decouples the sender of the request and its receivers,
- Simplifies your object because it doesn't have to know the chain's structure and keep direct reference to its members,
- Allows you to add or remove responsibilities dynamically by changing the members or the order of the chain.

Uses:

- Commonly used in Windows systems to handle events like mouse clicks and keyboard events.
- In java it is used in handling chain of Exceptions

Drawbacks:

- Execution of the request isn't guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage),
- Can be hard to observe the runtime characteristics and debug.

References

Text Reference

Design Patterns: Elements of Reusable Object-Oriented Software, GOF

Web Reference

https://www.cs.uah.edu/~rcoleman/CS307/DesignPatterns/DP17-ChainOfResponsibility.html
https://refactoring.guru/design-patterns
https://thetechstack.net/chain-of-responsibility-design-pattern/

THANK YOU

Prof. Vinay Joshi

Department of Computer Science and Engineering

vinayj@pes.edu

Object Oriented Analysis and Design with Java

UE21CS352B

Prof. Priya Badarinath
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352B: Object Oriented Analysis and Design with Java

Anti-Patterns - Architecture and Design Anti-Patterns

Prof. Priya Badarinath
Department of Computer Science and Engineering

Definitions

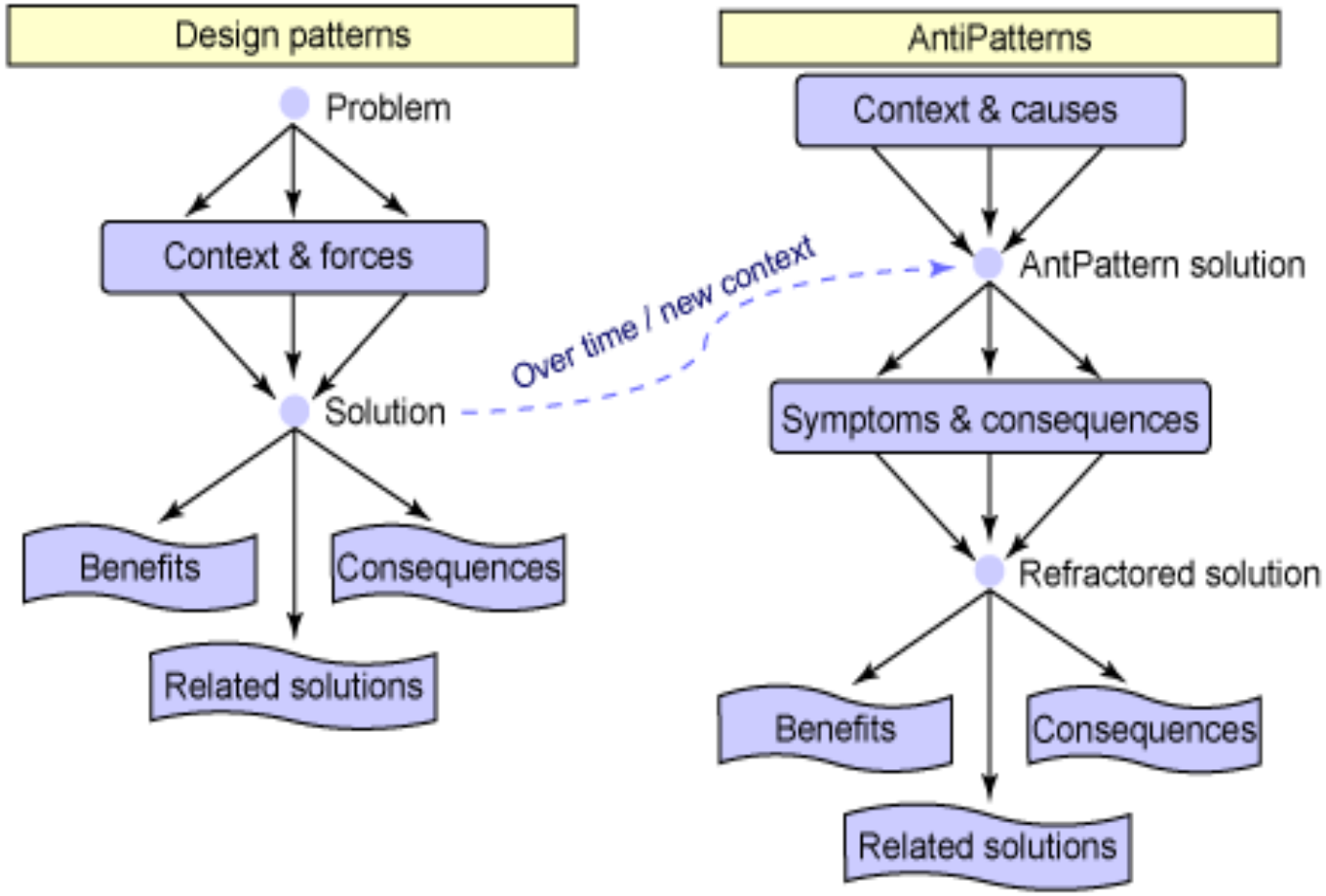
- Pattern: Good ideas
- Antipattern: Bad ideas
- Refactoring: Better ideas

What is an Antipattern?

- An AntiPattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences.
- The AntiPattern may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type of problem, or having applied a perfectly good pattern in the wrong context.

Antipatterns Vs. Design Patterns

- **Design Pattern-** a general repeatable solution to a commonly occurring problem
- **Antipattern-** Such a solution which is recognized as a poor way to solve the problem, and a refactored solution



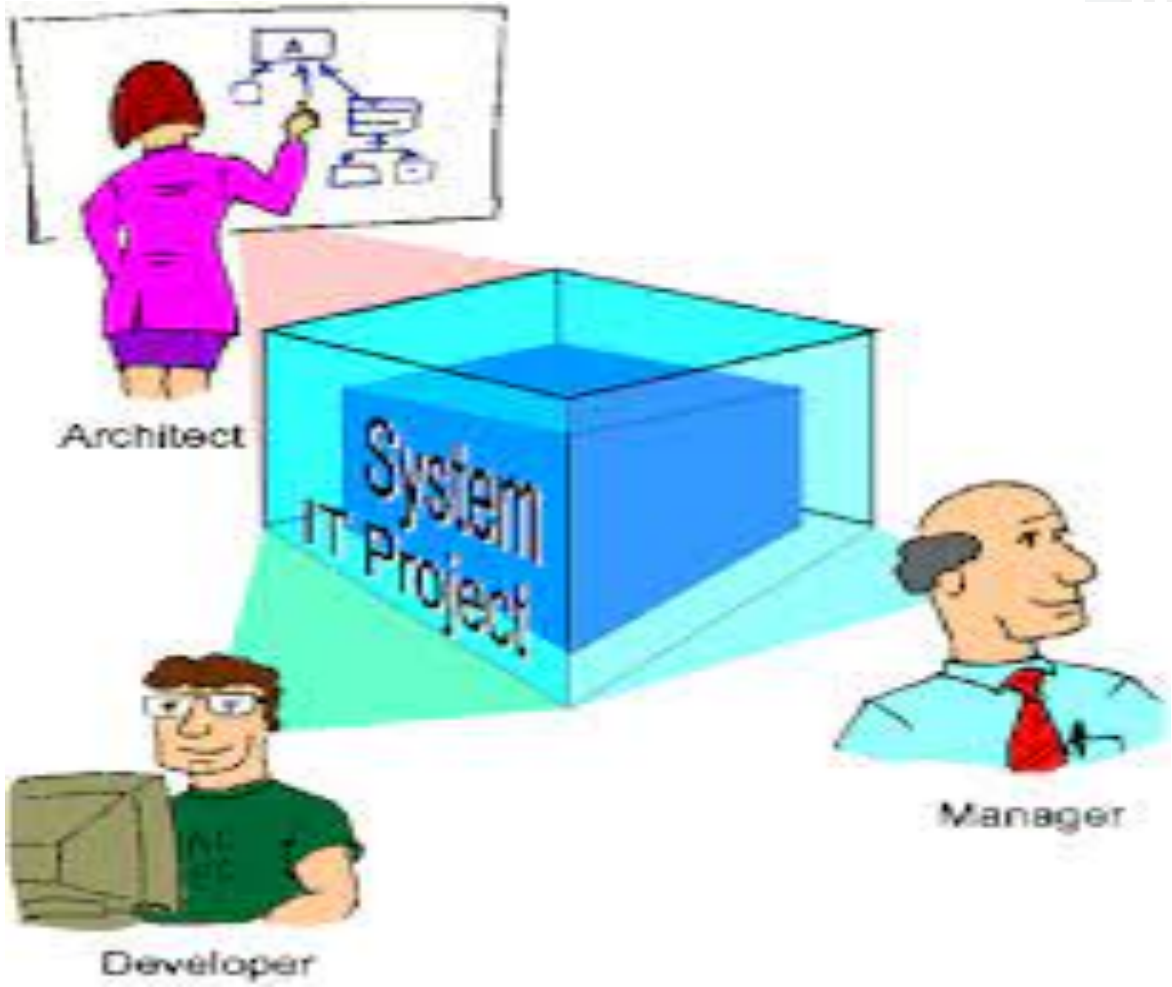
Why study Antipatterns?

- Antipatterns provide easily identifiable templates for common problems, as well as a path of action to rectify these problems.
- Antipatterns provide real world experience in recognizing recurring problems in the software industry providing a detailed remedy for the most common ones.
- Antipatterns provide a common vocabulary for identifying problems and discussing solutions.
- Antipatterns provide stress release in the form of shared misery.
- Antipatterns ensure common problems are not continually repeated within an organization

Viewpoints

AntiPatterns from three major viewpoints:

- The software developer,
- The software architect,
- The software manager



Principal AntiPattern viewpoints

Viewpoints

Development AntiPatterns describe situations encountered by the programmer when solving programming problems.

Architectural AntiPatterns focus on common problems in system structure, their consequences, and solutions. Many of the most serious unresolved problems in software systems occur from this perspective.

Management AntiPatterns describe common problems and solutions due to the software organization. It affects people in all software roles, and their solutions directly affect the technical success of the project.

Reference Model

A reference model for terminology common to all three viewpoints. The reference model is based upon three topics that introduce the key concepts of AntiPatterns:

- Root causes
  - Provide fundamental context for the AntiPattern
- Primal forces
  - are the key motivators for decision making
- Software design-level model (SDLM)
  - Define architectural scale

Reference Model – Root Causes

Root Causes

- Root causes are mistakes in software development that result in failed projects, cost overruns, schedule slips, and unfulfilled business needs.
- The root causes are based upon the "seven deadly sins," a popular analogy that has been used successfully to identify ineffective practices.

  Haste: Tight deadlines often lead to neglecting important activities

  Apathy: The attitude of not caring about solving known problems.

  Narrow-Mindedness: Refusal of developers to learn proven solutions.

Root Causes (Contd.,)

Sloth: adaptation of the most simple "solution".

Avarice: Greed in creating a system can result in very complex, and difficult to maintain software.

Ignorance: The lack of motivation to understand things.

Pride: The Failure to reuse existing software packages because they were not invented by a specific company

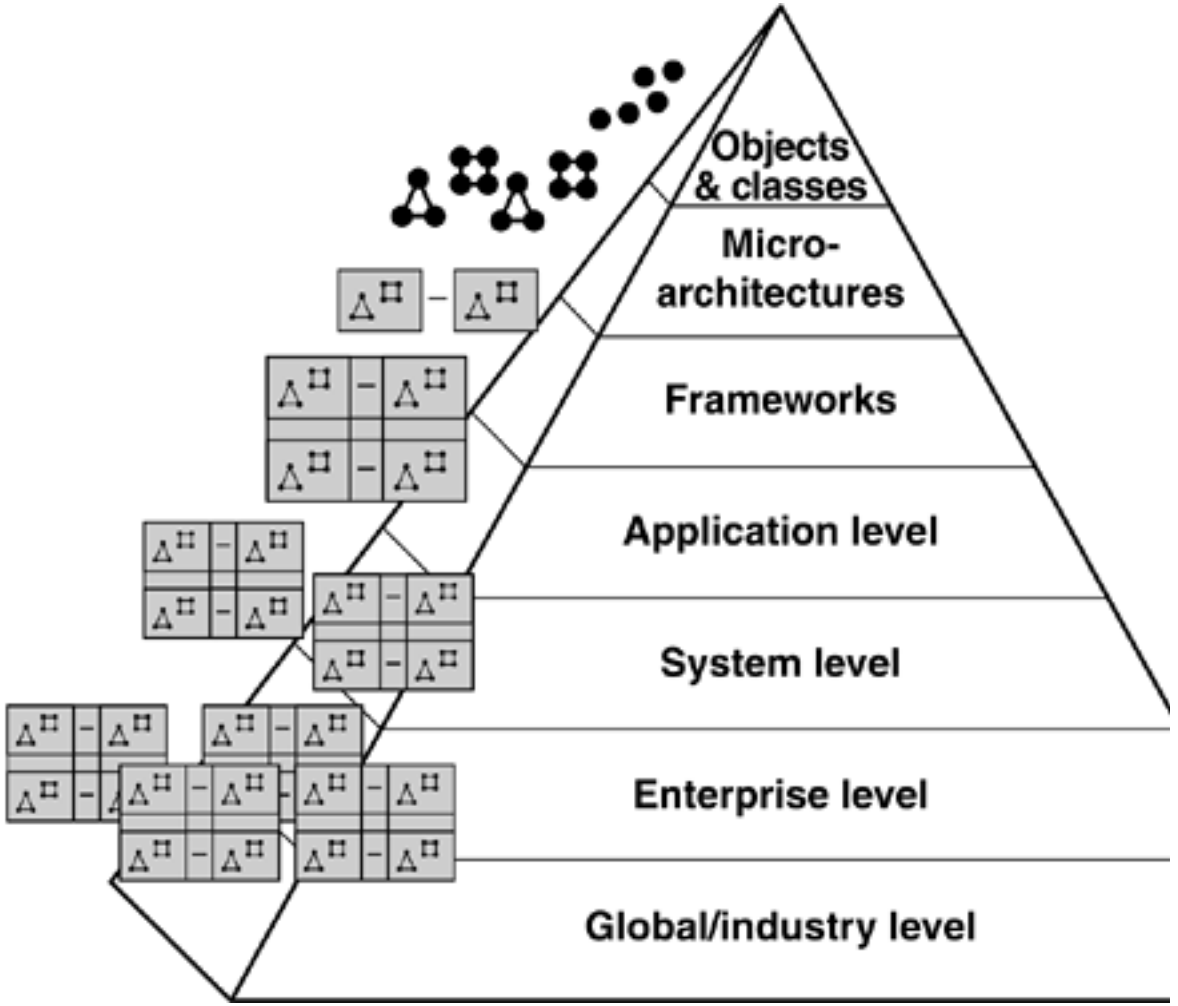
Primal Forces

Forces are concerns or issues that exist within a decision–making context. In a design solution, forces that are successfully addressed (or resolved) lead to benefits, and forces that are unresolved lead to consequences.

The primal forces include:

- Management of Functionality - Meeting the requirements
- Management of Performance - Meeting required speed and operation
- Management of Complexity - Defining abstractions
- Management of Change - Controlling the evolution of software
- Management of IT Resources - Managing people and IT artifacts
- Management of Technology Transfer - Controlling technology evolution

Software Design – Level Model



Software Design – Level Model (Contd.,)

- **The global level** contains the design issues that are globally applicable across all systems. This level is concerned with coordination across all organizations, which participate in cross−organizational communications and information sharing.
- **The enterprise level** is focused upon coordination and communication across a single organization. The organization can be distributed across many locations and heterogeneous hardware and software systems.
- **The system level** deals with communications and coordination across applications and sets of applications.
- **The application level** is focused upon the organization of applications developed to meet a set of user requirements. The macro−component levels are focused on the organization and development of application frameworks.

Software Design – Level Model (Contd.,)

- **The micro−component level** is centered on the development of software components that solve recurring software problems. Each solution is relatively self−contained and often solves just part of an even larger problem.
- **The object level** is concerned with the development of reusable objects and classes. The object level is more concerned with code reuse than design reuse. Each of the levels is discussed in detail along with an overview of the patterns documented at each level.

References

- https://sourcemaking.com/antipatterns/
- Antipatterns-Refactoring-Software-Architectures-and-Proj.pdf

THANK YOU

Department of Computer Science and Engineering

Object Oriented Analysis and Design with Java

UE21CS352B

Prof. Priya Badarinath
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352B: Object Oriented Analysis and Design with Java

Anti-Patterns - Architecture and Design Anti-Patterns

Department of Computer Science and Engineering

Project Management AntiPatterns

In the modern engineering profession, more than half of the job involves human communication and resolving people issues. The management AntiPatterns identify some of the key scenarios in which these issues are destructive to software processes.

The areas where managers play vital role,

1. Software process management
2. Resource management (human & IT infrastructure)
3. External relationship management (e.g., customers, development partners)

Project Management AntiPatterns – Analysis Paralysis

Symptoms and Consequences:

- There are multiple project restarts and much model rework, due to personnel changes or changes in project direction.
- Design and implementation issues are continually reintroduced in the analysis phase.
- Cost of analysis exceeds expectation without a predictable end point.
- The analysis phase no longer involves user interaction. Much of the analysis performed is speculative.
- The complexity of the analysis models results in intricate implementations, making the system difficult to develop, document, and test.
- Design and implementation decisions such as those used in the Gang of Four design patterns are made in the analysis phase.

Project Management AntiPatterns – Analysis Paralysis

Typical Causes:

- The management process assumes a waterfall progression of phases. In reality, virtually all systems are built incrementally even if not acknowledged in the formal process.
- Management has more confidence in their ability to analyze and decompose the problem than to design and implement.
- Management insists on completing all analysis before the design phase begins.
- Goals in the analysis phase are not well defined.
- Planning or leadership lapses when moving past the analysis phase.
- Management is unwilling to make firm decisions about when parts of the domain are sufficiently described.
- The project vision and focus on the goal/deliverable to customer is diffused. Analysis goes beyond providing meaningful value.

Project Management AntiPatterns – Analysis Paralysis

Refactored Solution:

- Key to the success of object-oriented development is incremental development. Whereas a waterfall process assumes a priori knowledge of the problem, incremental development processes assume that details of the problem and its solution will be learned in the course of the development process.

- There are two kinds of **increments**: **internal and external**. An **internal increment** builds software that is essential to the infrastructure of the implementation. For example, a third-tier database and data-access layer would comprise an internal increment. Internal increments build a common infrastructure that is utilized by multiple use cases. In general, internal increments minimize rework. An **external increment** comprises user-visible functionality.

Software Architecture Antipatterns

- Architecture AntiPatterns focus on the system-level and enterprise-level structure of applications and components.
- Although the engineering discipline of software architecture is relatively immature, what has been determined repeatedly by software research and experience is the overarching importance of architecture in software development.

The following AntiPatterns focus on some common problems and mistakes in the creation, implementation, and management of architecture.

Software Architecture Antipatterns – Vendor Lock-In

Vendor Lock-In

- A software project adopts a product technology and becomes completely dependent upon the vendor's implementation. When upgrades are done, software changes and interoperability problems occur, and continuous maintenance is required to keep the system running.

- In addition, expected new product features are often delayed, causing schedule slips and an inability to complete desired application software features.

Software Architecture Antipatterns – Vendor Lock-In

Vendor Lock-In – Symptoms and Consequences

- Commercial product upgrades drive the application software maintenance cycle.
- Promised product features are delayed or never delivered, subsequently, causing failure to deliver application updates.
- The product varies significantly from the advertised open systems standard.
- If a product upgrade is missed entirely, a product repurchase and reintegration is often necessary.

Software Architecture Antipatterns – Vendor Lock-In

Vendor Lock-In – Typical Causes

- The product varies from published open system standards because there is no effective conformance process for the standard.
- The product is selected based entirely upon marketing and sales information, and not upon more detailed technical inspection.
- There is no technical approach for isolating application software from direct dependency upon the product.
- Application programming requires in-depth product knowledge.
- The complexity and generality of the product technology greatly exceeds that of the application needs; direct dependence upon the product results in failure to manage the complexity of the application system architecture.

Software Architecture Antipatterns – Vendor Lock-In

Vendor Lock-In – Refactored Solution

The refactored solution to the Vendor Lock-In AntiPattern is called *isolation layer.* An isolation layer separates software packages and technology. This solution is applicable when one or more of the following conditions apply:

- *Isolation of application software from lower-level infrastructure.* This infrastructure may include middleware, operating systems, security mechanisms, or other low-level mechanisms.
- *Changes to the underlying infrastructure are anticipated within the life cycle of the affected software;* for example, new product releases or planned migration to new infrastructure.
- *A more convenient programming interface is useful or necessary.* The level of abstraction provided by the infrastructure is either too primitive or too flexible for the intended applications and systems.
- *There is a need for consistent handling of the infrastructure across many systems.* Some heavyweight conventions for default handling of infrastructure interfaces must be instituted.
- Multiple infrastructures must be supported, either during the life cycle or concurrently.

Software Development Antipatterns

Good software structure is essential for system extension and maintenance. Software development is a chaotic activity, therefore the implemented structure of systems tends to stray from the planned structure as determined by architecture, analysis, and design.

Software Development Antipatterns – The Blob

The Blob

Procedural-style design leads to one object with a lion's share of the responsibilities, while most other objects only hold data or execute simple processes. The solution includes refactoring the design to distribute responsibilities more uniformly and isolating the effect of changes.

Background:

The Blob is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data. This AntiPattern is characterized by a class diagram composed of a single complex controller class surrounded by simple data classes. The key problem here is that the majority of the responsibilities are allocated to a single class

Software Development Antipatterns – The Blob

Symptoms and Consequences

- Single class with a large number of attributes, operations, or both. A class with 60 or more attributes and operations usually indicates the presence of the Blob.
- A disparate collection of unrelated attributes and operations encapsulated in a single class. An overall lack of cohesiveness of the attributes and operations is typical of the Blob.
- The Blob Class is typically too complex for reuse and testing. It may be inefficient, or introduce excessive complexity to reuse the Blob for subsets of its functionality.
- The Blob Class may be expensive to load into memory, using excessive resources, even for simple operations.

Software Development Antipatterns – The Blob

Typical Causes

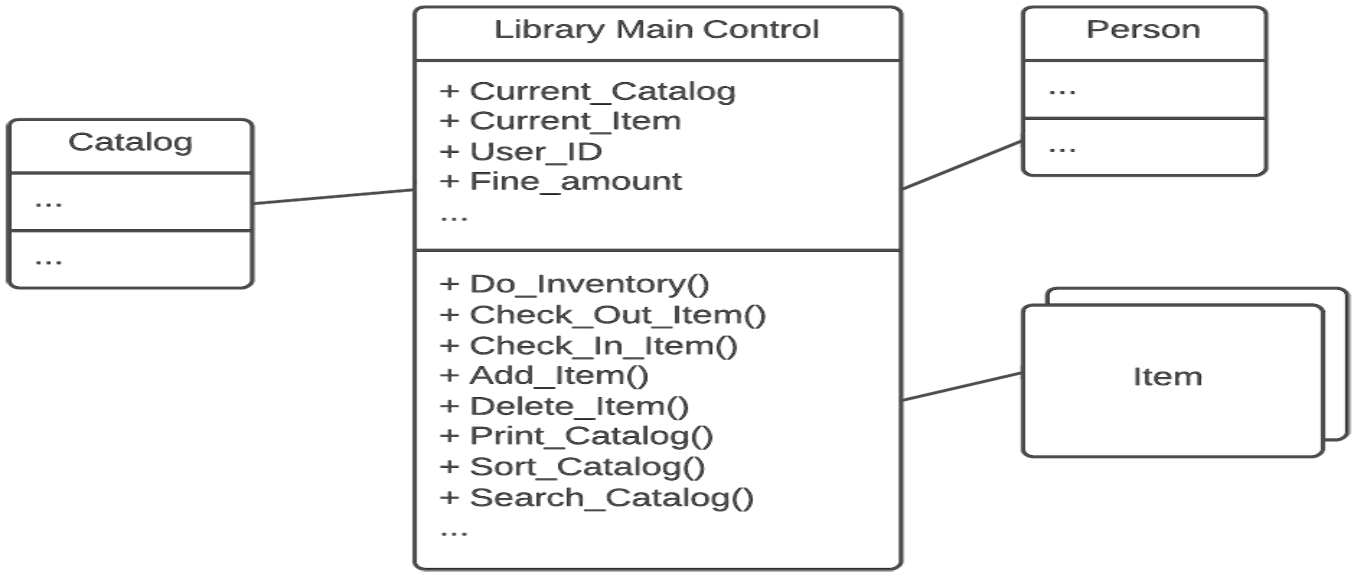
- Lack of an object-oriented architecture.
- Lack of (any) architecture.
- Lack of architecture enforcement.
- Too limited intervention.
- Specified disaster.

Software Development Antipatterns – The Blob

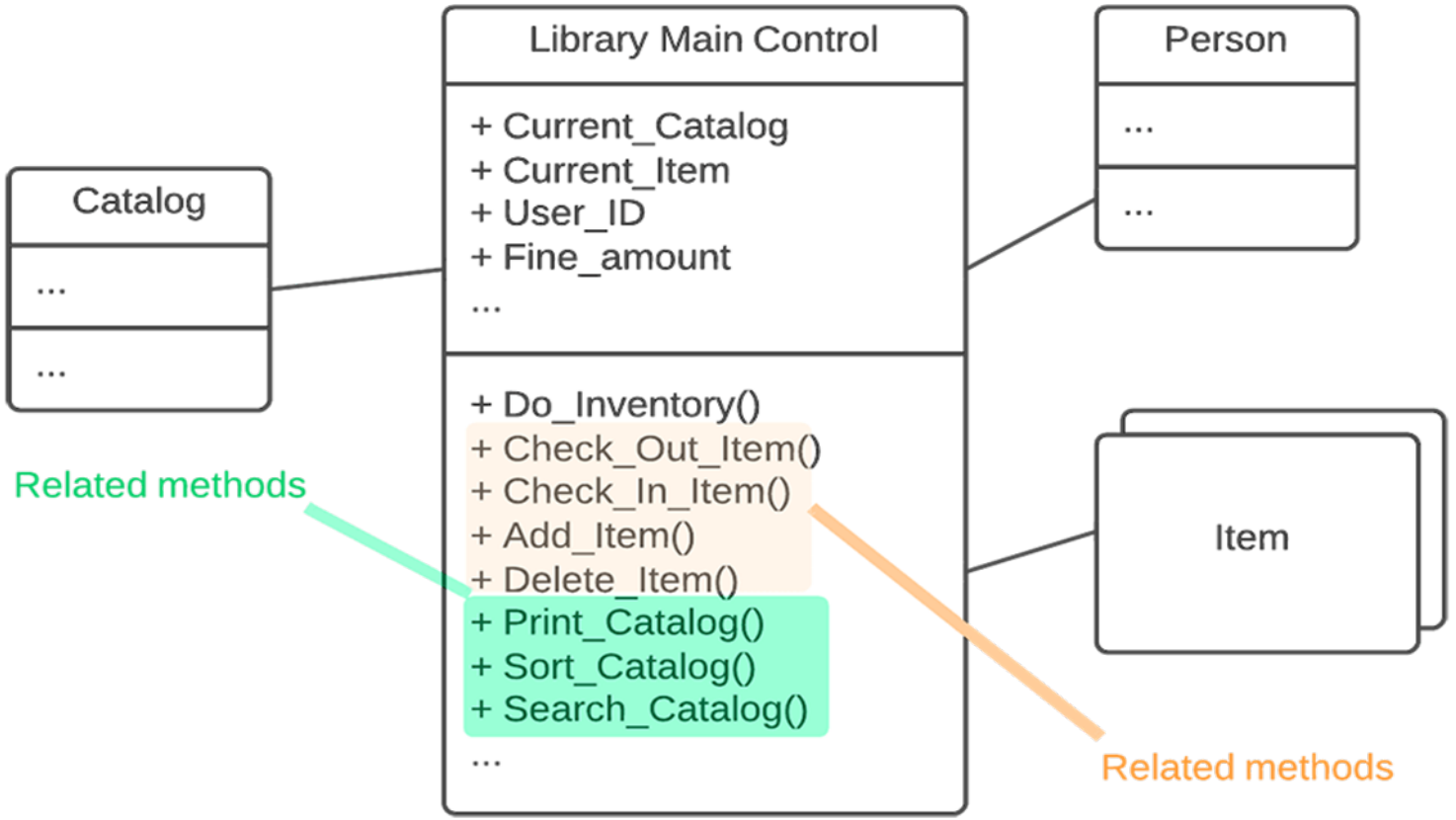
Refactored Solution

Step 1:

- Identify or categorize related attributes and operations according to contracts. These contracts should be cohesive in that they all directly relate to a common focus, behavior, or function within the overall system.



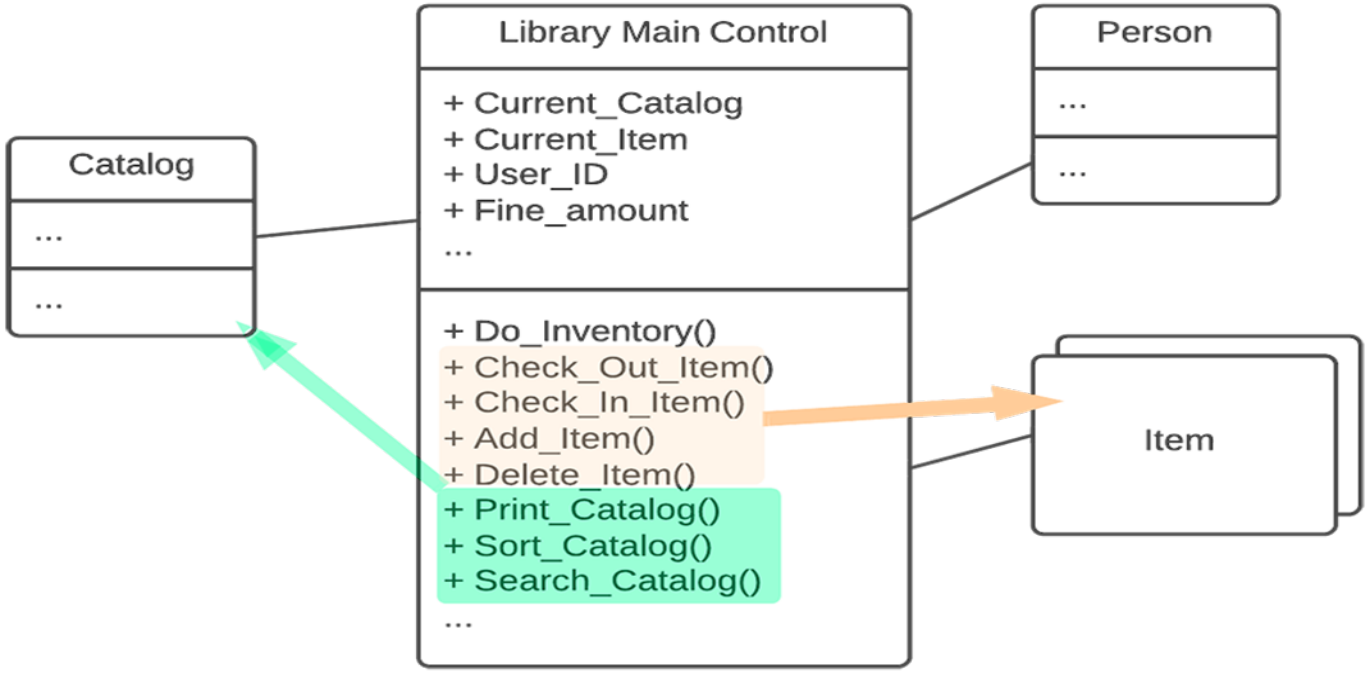
Software Development Antipatterns – The Blob



Software Development Antipatterns – The Blob

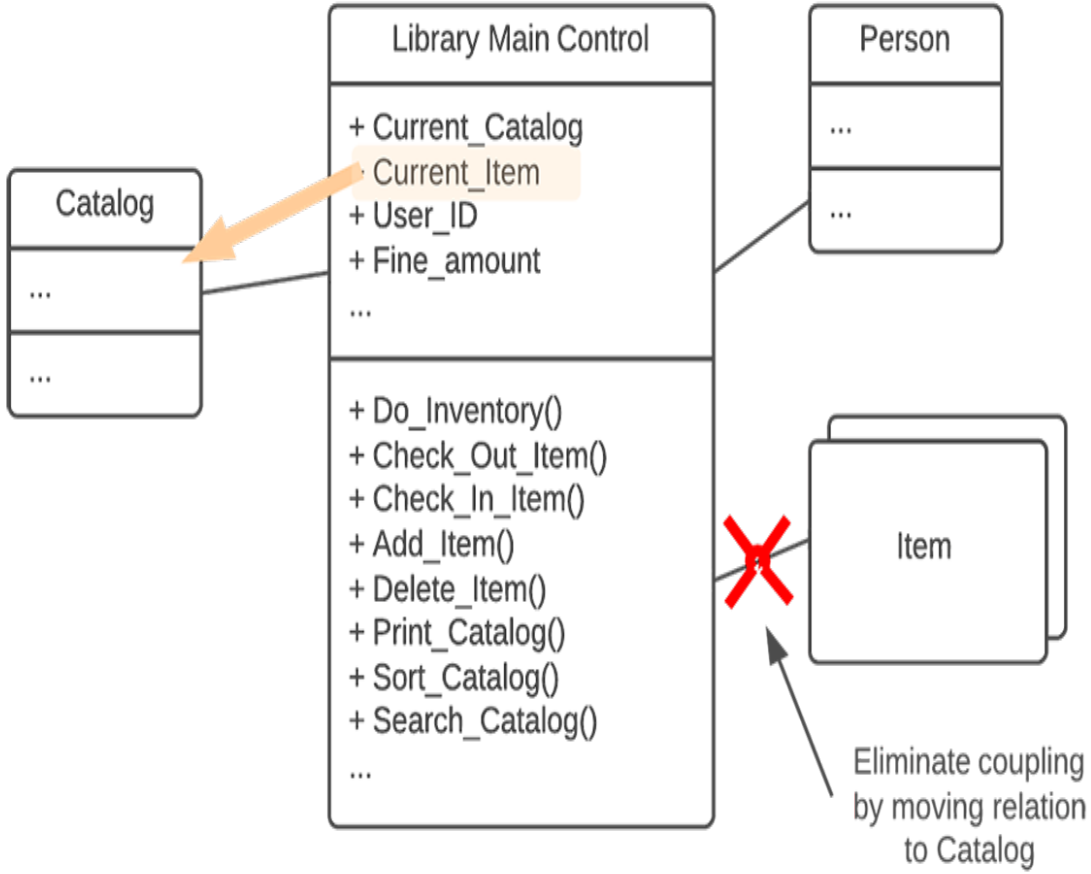
Step 2:

- Look for "natural homes" for these contract-based collections of functionality and then migrate them there. In this example, we gather operations related to catalogs and migrate them from the LIBRARY class and move them to the CATALOG class.



Software Development Antipatterns – The Blob

- The third step is to remove all "far-coupled," or redundant, indirect associations. In the example, the ITEM class is initially far-coupled to the LIBRARY class in that each item really belongs to a CATALOG, which in turn belongs to a LIBRARY.
- Next, where appropriate, we migrate associates to derived classes to a common base class. In the example, once the far-coupling has been removed between the LIBRARY and ITEM classes, we need to migrate ITEMs to CATALOGs,
- Finally, we remove all transient associations, replacing them as appropriate with type specifiers to attributes and operations arguments.



References

- https://sourcemaking.com/antipatterns/
- Antipatterns-Refactoring-Software-Architectures-and-Proj.pdf

THANK YOU

Department of Computer Science and Engineering

Object Oriented Analysis and Design with Java

UE21CS352B

Prof. Priya Badrainath
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352B : Object Oriented Analysis and Design with Java

Unit 4
Structural Patterns – Adapter

Prof. Priya Badarinath

Department of Computer Science and Engineering

Agenda

- ❑ Introduction to Structural design patterns
- ❑ Types of Structural Design Patterns.
- ❑ Adapter-definition
  - ✔ Motivation
  - ✔ Intent
  - ✔ Implementation
  - ✔ Applicability
  - ✔ Structure-Consequence
  - ✔ Issues

Introduction to Structural Design Pattern

- Structural design patterns are concerned with how classes and objects can be composed, to form larger structures.
- The structural design patterns simplifies the structure by identifying the relationships.
- These patterns focus on, how the classes inherit from each other and how they are composed from other classes.
- Structural *class* patterns use inheritance to compose interfaces or implementations.
- Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

Types of Structural Design pattern: Scope

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Adapter: Class, Object Structural

Motivation

- The adapter pattern is adapting between classes and objects.
- Like any adapter in the real world it is used to be an interface, a bridge between two objects.
  Ex: In real world we have adapters for power supplies, adapters for camera memory cards, and so on.
- Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Advantage of Adapter Pattern

- It allows two or more previously incompatible objects to interact.
- It allows reusability of existing functionality.

Adapter: Class, Object Structural

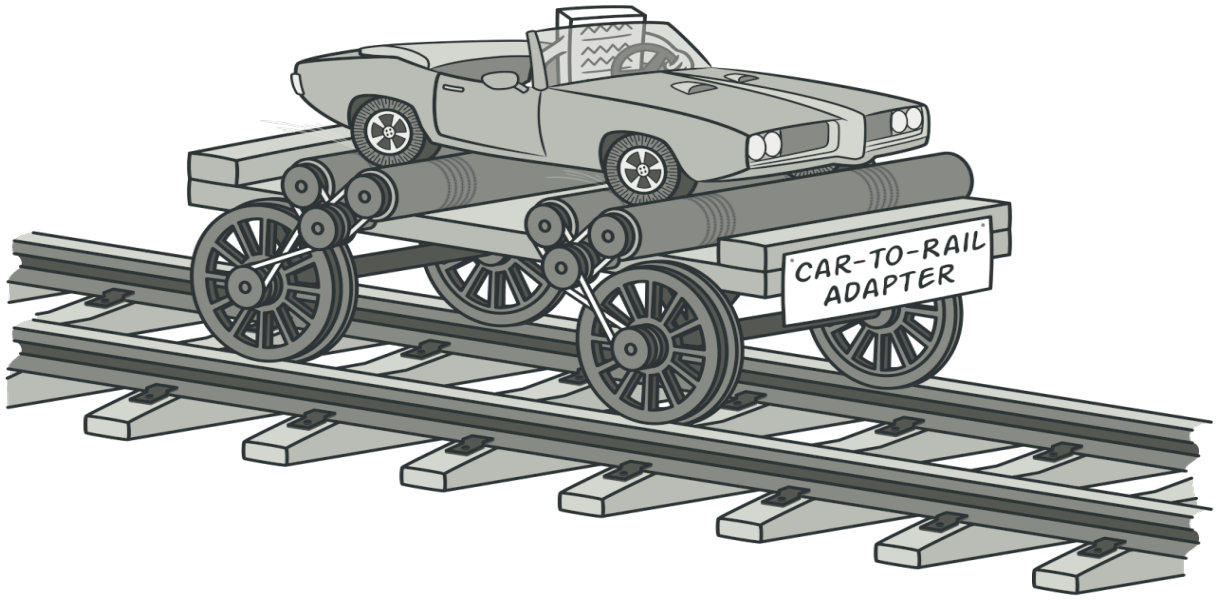
Intent

- Adapter lets classes work together that couldn' t otherwise because of incompatible interfaces.

- An Adapter Pattern says that "convert the interface of a class into another interface that a client wants". In other words, to provide the interface according to client requirement while using the services of a class with a different interface.

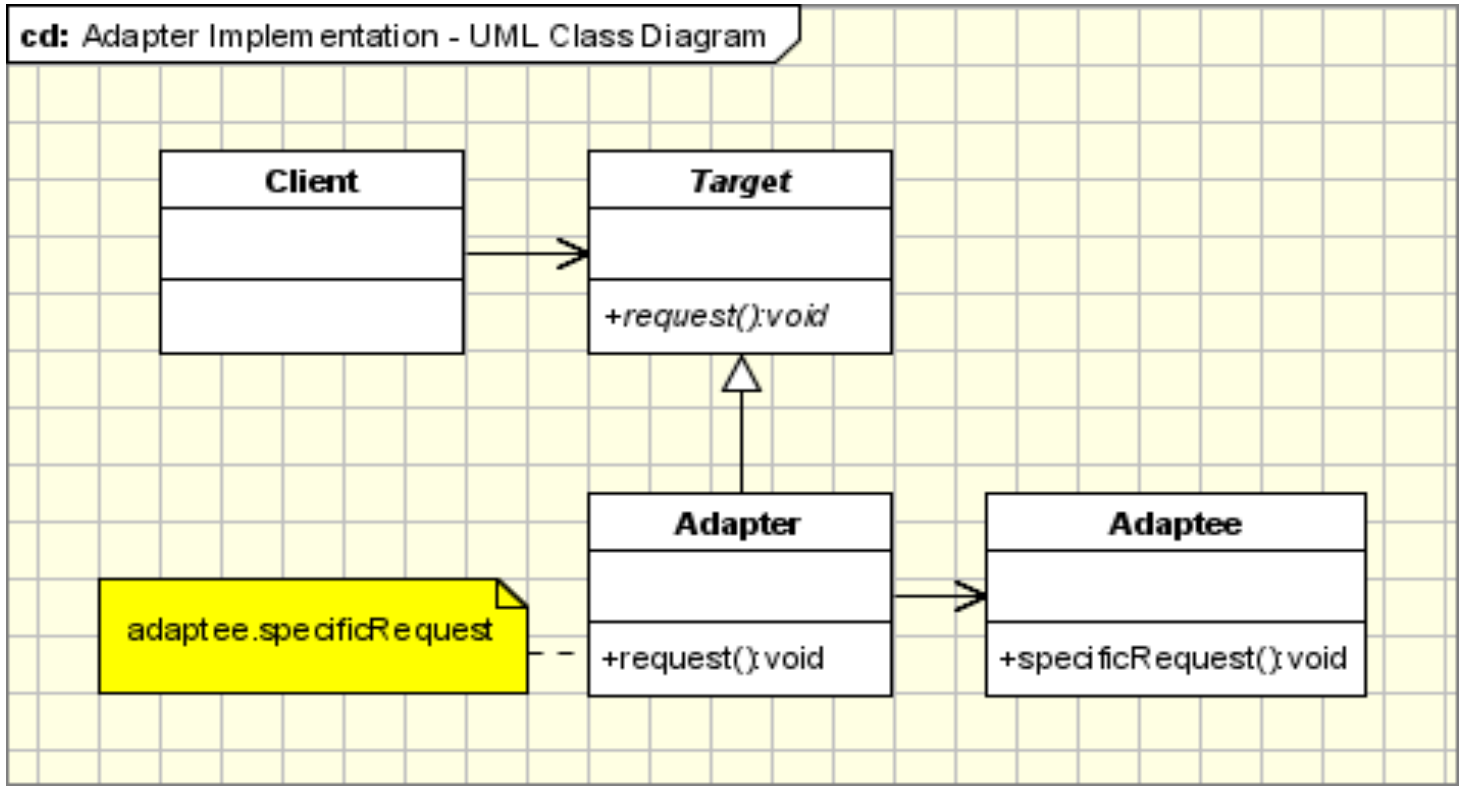
- The Adapter Pattern is also known as Wrapper.

Adapter is a structural design pattern that allows classes with incompatible interfaces to collaborate.



Adapter: Implementation

UML class diagram for the Adapter Pattern



The classes/objects participating in adapter pattern:

1. Target - defines the domain-specific interface that Client uses.
2. Adapter - adapts the interface Adaptee to the Target interface.
3. Adaptee - defines an existing interface that needs adapting.
4. Client - collaborates with objects conforming to the Target interface.

Adapter class: This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.

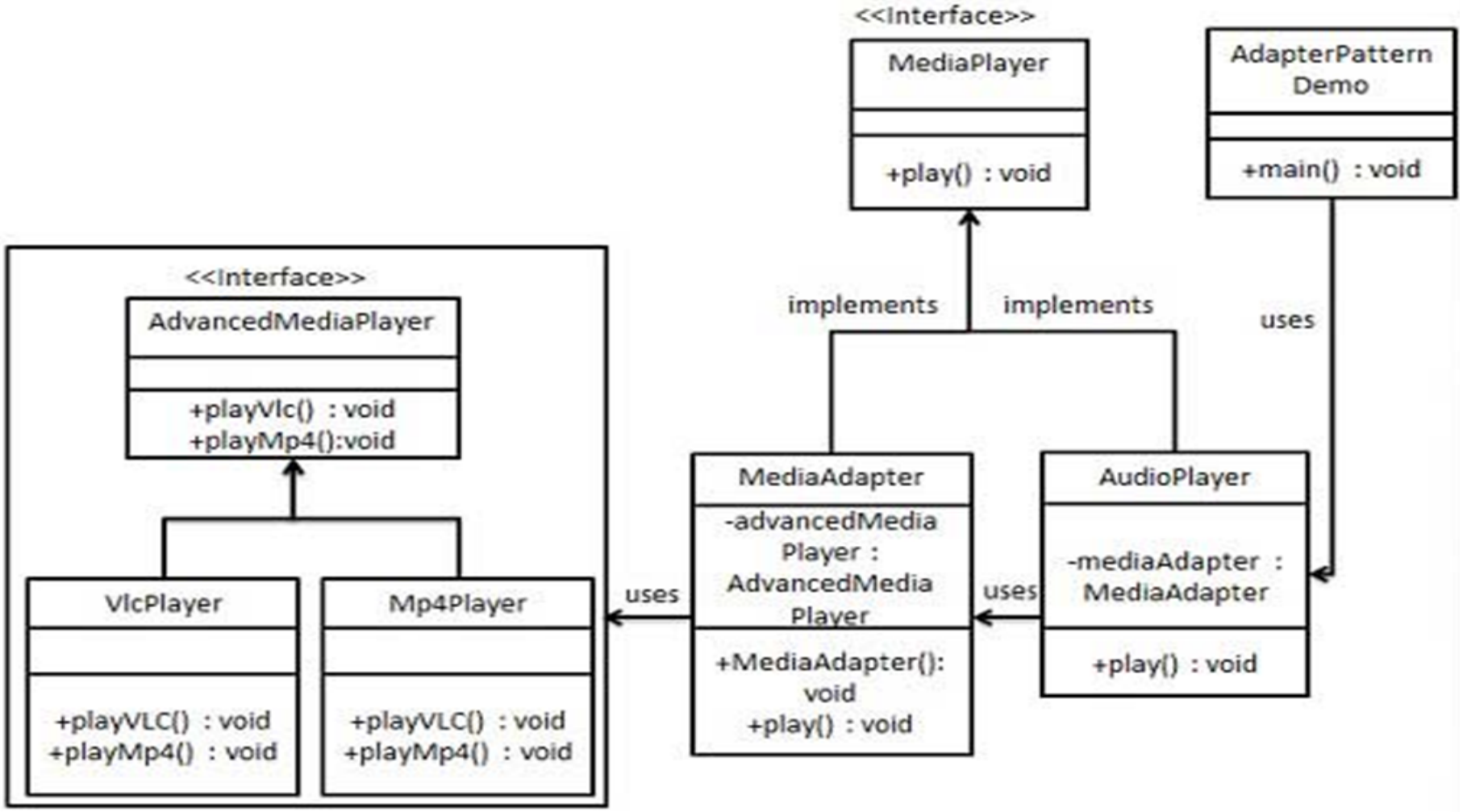
Adapter: Implementation example-1

Problem Statement: We have a MediaPlayer interface and a concrete class AudioPlayer implementing the MediaPlayer interface. AudioPlayer can play mp3 format audio files by default. We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface. These classes can play vlc and mp4 format files. We want to make AudioPlayer to play other formats as well.

Solution: To attain this, we have created an adapter class MediaAdapter which implements the MediaPlayer interface and uses AdvancedMediaPlayer objects to play the required format. AudioPlayer uses the adapter class MediaAdapter passing it the desired audio type without knowing the actual class which can play the desired format. AdapterPatternDemo, our demo class will use AudioPlayer class to play various formats.

Demo….>refer to AdapterPatternDemo.java

Design Solution



Design Solution

Adapter Pattern

Modules :

Module Type	Module Example
Adapter (concrete of AbstractA)	*MediaAdapter*
Adaptee (concrete of AbstractA)	*AudioPlayer*
AbstractB	*AdvanceMediaPlayer*
ConcreteB	*VLCplayer, MP4player*
AbstactA	*MediaPlayer*
Demo	*AdapterPatternDemo*

Adapter: Scenario example-2

Suppose you have a Bird class with fly() and makeSound()methods. And also a ToyDuck class with squeak() method. Let's assume that you are short on ToyDuck objects and you would like to use Bird objects in their place. Birds have some similar functionality but implement a different interface, so we can't use them directly.

Solution: So, we will use adapter pattern. Here our client would be ToyDuck and adaptee would be Bird.(refer to demo main.java

The adapter pattern we have implemented above is called Object Adapter Pattern because the adapter holds an instance of adaptee.
What a Object adapter??? Next slides we will explain in detail

Adapter: Scenario example-2

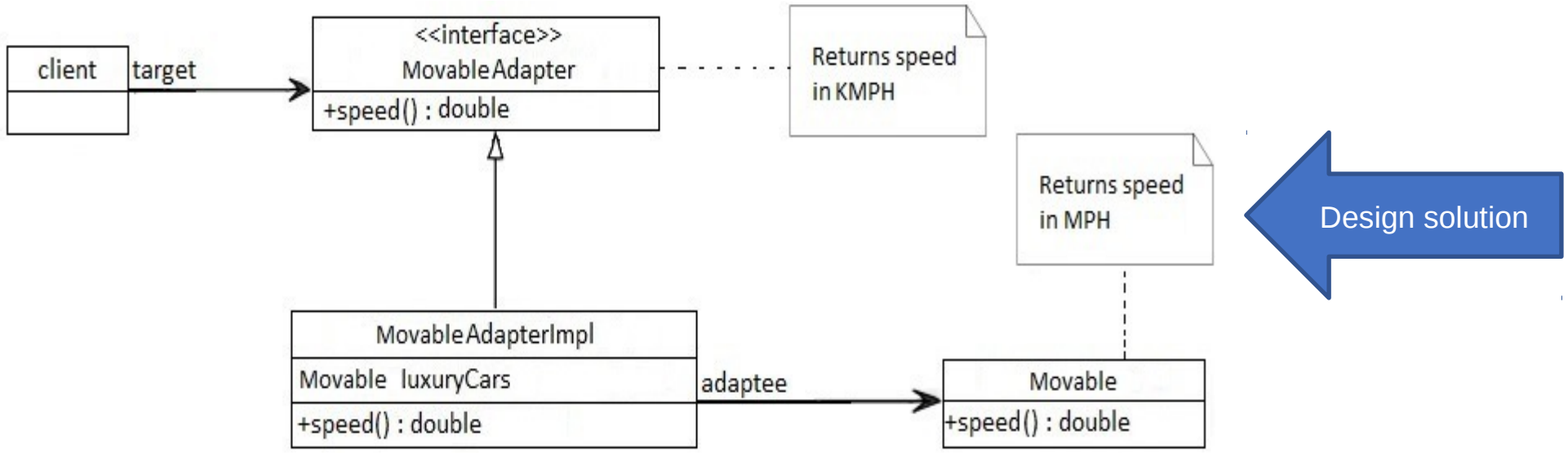
Explain: Here we have created a BirdAdapter class implementing inteface ToyDuck to convert Bird 's makeSound() method to makeSound() 's squeak() ToyDuck . This way, we can still call the squeak() of Bird objects through BirdAdapter

Suppose we have a bird that can makeSound(), and we have a plastic toy duck that can squeak(). Now suppose our client changes the requirement and he wants the toyDuck to makeSound then ?

Simple solution is that we will just change the implementation class to the new adapter class and tell the client to pass the instance of the bird(which wants to squeak()) to that class.
Before : ToyDuck toyDuck = new PlasticToyDuck();
After : ToyDuck toyDuck = new BirdAdapter(sparrow);
You can see that by changing just one line the toyDuck can now do Chirp Chirp !!

Adapter: Scenario example-3

Consider a scenario in which there is an app that's developed in the US which returns the top speed of luxury cars in miles per hour (MPH). Now we need to use the same app for our client in the UK that wants the same results but in kilometers per hour (km/h).
To deal with this problem, we'll create an adapter which will convert the values and give us the desired results:



(Students can Try coding with the given design solution)

Applicability

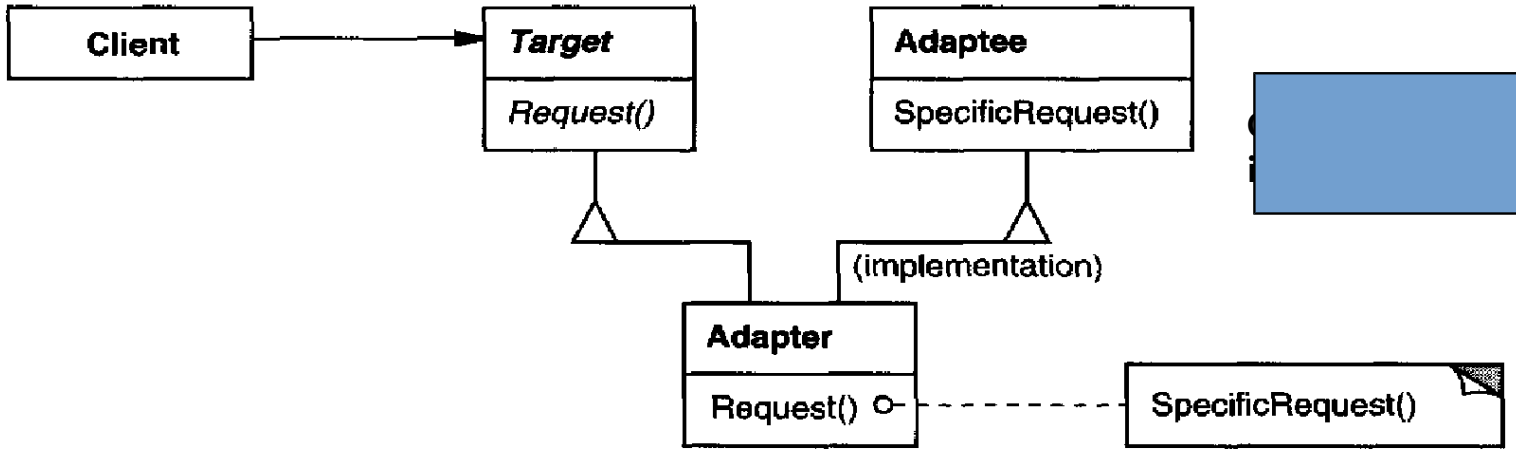
Use the Adapter pattern when

- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- (object adapter only) you need to use several existing subclasses, but it's unpractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

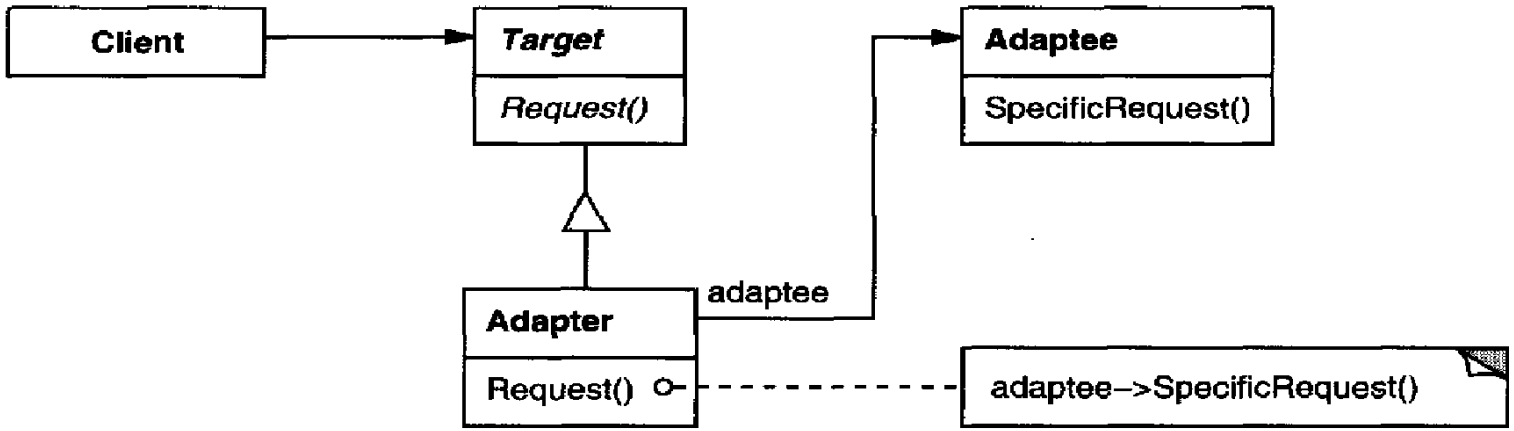
Object Oriented Analysis and Design with Java

Structure

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



An object adapter relies on object composition: Based on delegation



Consequence

Class and object adapters have different trade-offs.

A class adapter

- adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
- let's Adapter override some of Adaptee’s behavior, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An object adapter

- **let's a single Adapter work with many Adaptees**—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Difference: Class and Object structure pattern

- Objects Adapters uses composition, the Adaptee delegates the calls to Adaptee (opossed to class adapters which extends the Adaptee).
- The main advantage is that the object Adapter adapts not only the Adaptee but all its subclasses. All it's subclasses with one "small" restriction: all the subclasses which don't add new methods, because the used mechanism is delegation. So for any new method the Adapter must be changed or extended to expose the new methods as well.
- The main disadvantage is that it requires to write all the code for delegating all the necessary requests to the Adaptee.

- Class adapter uses inheritance instead of composition. It means that instead of delegating the calls to the Adaptee, it subclasses it. In conclusion it must subclass both the Target and the Adaptee.
- There are advantages and disadvantages: It adapts the specific Adaptee class. The class it extends. If that one is subclassed it can not be adapted by the existing adapter.
- It doesn't require all the code required for delegation, which must be written for an Object Adapter.
- If the Target is represented by an interface instead of a class then we can talk about "class" adapters, because we can implement as many interfaces as we want.

Issues to consider when using the Adapter pattern

❑ *How Much the Adapter Should Do?*

It should do how much it has to in order to adapt. It's very simple, if the Target and Adaptee are similar then the adapter has just to delegate the requests from the Target to the Adaptee. If Target and Adaptee are not similar, then the adapter might have to convert the data structures between those and to implement the operations required by the Target but not implemented by the Adaptee.

❑ *Using two-way adapters to provide transparency.*

A potential problem with adapters is that they aren't transparent to all clients. An adapted object no longer conforms to the Adaptee interface, so it can't be used as is wherever an Adaptee object can. Two-way adapters can provide such transparency. Specifically, they're useful when two different clients need to view an object differently.

Link for adapter_program

References

Text Reference

- Design Patterns: Elements of Reusable Object-Oriented Software, GOF

Web Reference

- https://www.javatpoint.com/adapter-pattern
- https://www.oodesign.com/adapter-pattern.html
- https://www.geeksforgeeks.org/adapter-pattern/
- https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm
- https://itzone.com.vn/en/article/adapter-design-pattern-in-java

THANK YOU

Prof. Priya Badarinath

Department of Computer Science and Engineering

priyab@pes.edu

Object Oriented Analysis and Design with Java

UE21CS352B

Prof. Priya Badarinath
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352B : Object Oriented Analysis and Design with Java

Structural Patterns : Facade pattern

Prof. Sindhu R Pai
Department of Computer Science and Engineering

Agenda

- What is façade?
- Why Façade?
- Pictorial Representation
- Applicability
- Advantages
- Known Uses
- Implementation
- References

What is facade?

Meaning:

- The principal front of a building, that faces on to a street or open space.

  "the house has a half-timbered façade"

- A deceptive outward appearance.

  "her flawless public façade masked private despair"

Facade in Software

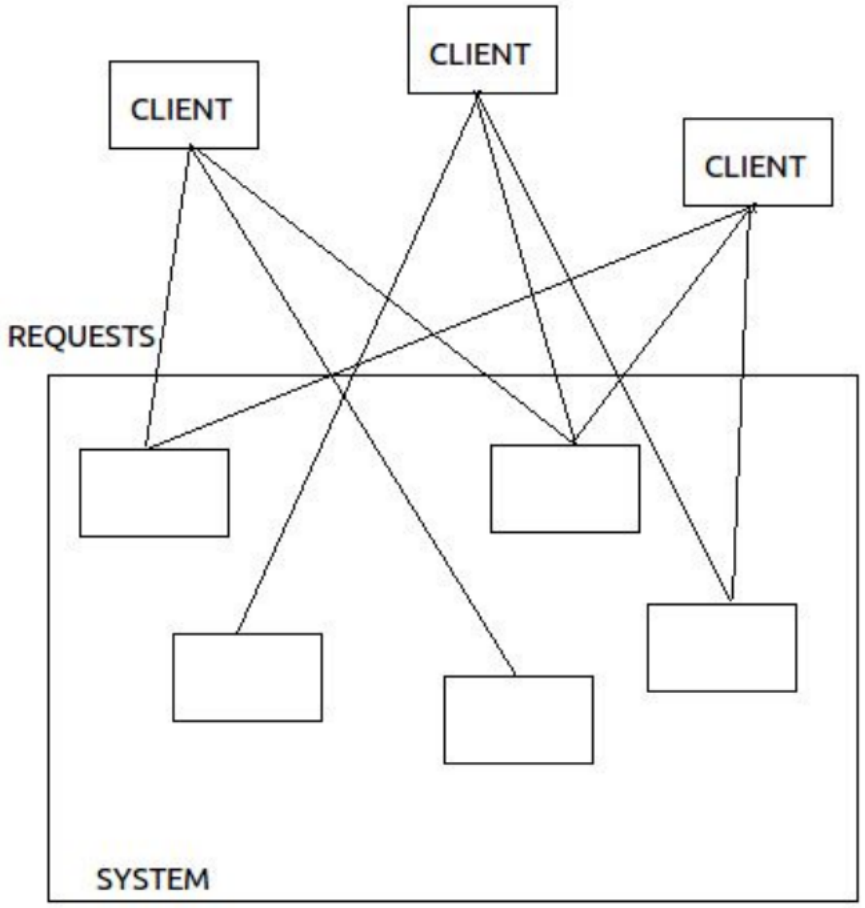
- An object that provides a **simplified interface to a larger body** of code, a library, a framework, or any other complex set of classes.
- The main intent of Façade is to provide a **unified interface to a set of interfaces in a subsystem**. Façade defines a **higher-level interface** that makes the subsystem easier to use.
- Example: fopen is an interface to open and read system commands.
- A structural design pattern, which **wraps a complicated subsystem with a simpler interface.**
- If the Facade is the only access point for the subsystem, it will limit the features and flexibility that **"power users**" may need

Why Facade?

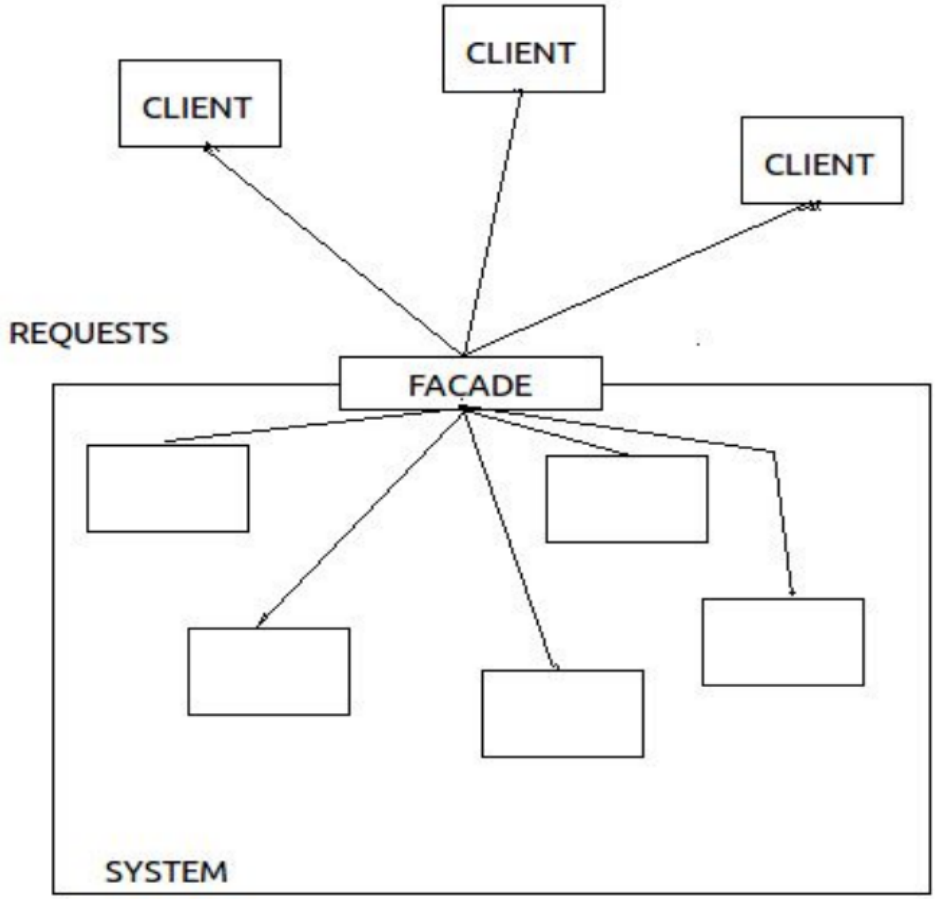
- Structuring a system into subsystems **helps reduce complexity**.

- A common design goal is to **minimize the communication and dependencies between subsystems**. One way to achieve this goal is to introduce a façade object that provides a single, simplified interface to the more general facilities of a subsystem.

Pictorial representation



Without facade



With facade

Applicability

- To provide a **simple interface to a complex subsystem**. A façade can provide **a simple default view of the subsystem** that is good enough for most clients.
- Introduce a façade to **decouple the subsystem from clients and other subsystems**, thereby promoting **subsystem independence and portability**.
- Use a façade to **define an entry point to each subsystem** level.
- To **simplify the dependencies between subsystems** by making them communicate with each other solely through their façades.

Advantages

- It **shields clients from subsystem components**, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
- It **promotes weak coupling between the subsystem and its clients**. Often the components in a subsystem are strongly coupled.
- This can **eliminate complex or circular dependencies**. This can be an important consequence when the client and the subsystem are implemented independently.
- Reducing compilation dependencies with façades **can limit the recompilation needed for a small change** in an important subsystem.

Known uses

- In the **ET++ application framework**, an application that can have built-in browsing tools for inspecting its objects at run-time. These browsing tools are implemented in a separate subsystem that includes a façade class called "ProgrammingEnvironment." This façade defines operations such as InspectObject and InspectClass for accessing the browsers.

- The **Choices operating system** uses façade to compose many frameworks into one. The key abstractions in Choices are processes, storage, and address spaces. For each of these abstractions there is a corresponding subsystem, implemented as a framework, that supports porting Choices to a variety of different hardware platforms.

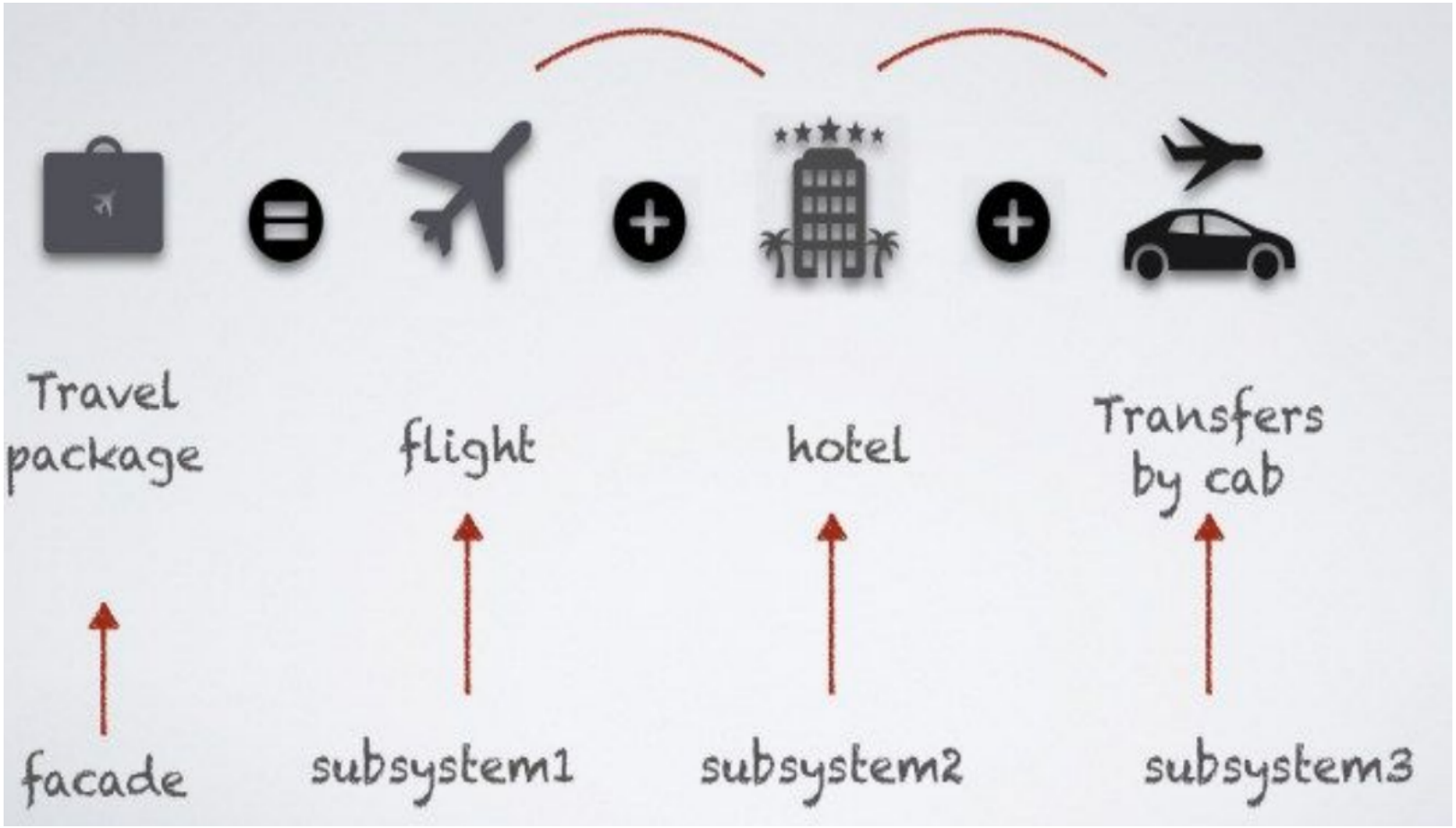
Implementation

• **Reduce client-subsystem coupling**. This can be done by simply **creating a "Façade Abstract Class" and concrete subclasses for the implementation of the subsystem**. Now the client class can communicate with the subsystem through the "**Abstract Façade Class**".

An alternative approach is to configure a façade object with different subsystem objects

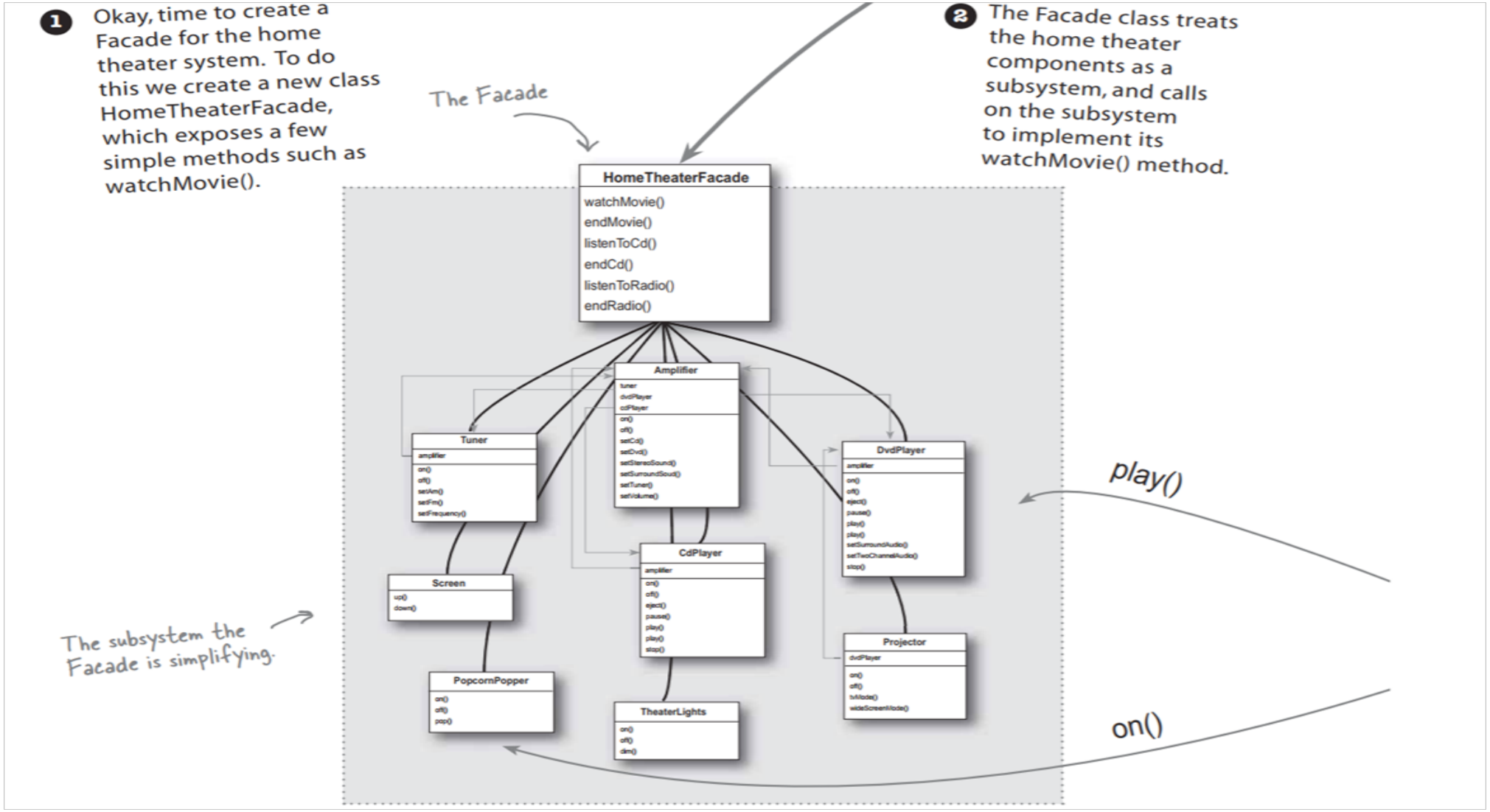
• Public versus Private Interfaces. Classes and Subsystems are similar. Both encapsulate something. Both have private and public interfaces. The **public interface consists of classes accessible by all Clients** whereas the private interface is only for subsystem extenders. **Façade is part of the public interface**

Example -1: Booking a package



Booking a package

Example -2: Designing Hometheater



References

- Facade Pattern from Head First Design Patterns (javaguides.net)
- Facade Design Pattern (sourcemaking.com)
- https://refactoring.guru/design-patterns/java

THANK YOU

Prof.Priya Badarinath

Department of Computer Science and Engineering

Object Oriented Analysis and Design with Java

UE21CS352B

Prof. Priya Badarinath
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352B : Object Oriented Analysis and Design with Java

Unit 4
Structural Patterns - Proxy

Department of Computer Science and Engineering

Proxy

- In proxy design pattern, a proxy object provide a surrogate or placeholder for another object to control access to it.
- A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Proxy design pattern - Intent

- Proxy is heavily used to implement lazy loading related usecases where we do not want to create full object until it is actually needed.
- Using the proxy pattern, a class represents the functionality of another class.

Proxy design pattern - Intent

In the Proxy Design Pattern, a client does not directly talk to the original object, it delegates call to the proxy object which calls the methods of the original object. Moreover, the important point is that the client does not know about the proxy. The proxy acts as an original object for the client. There are three main variations of the Proxy Pattern:

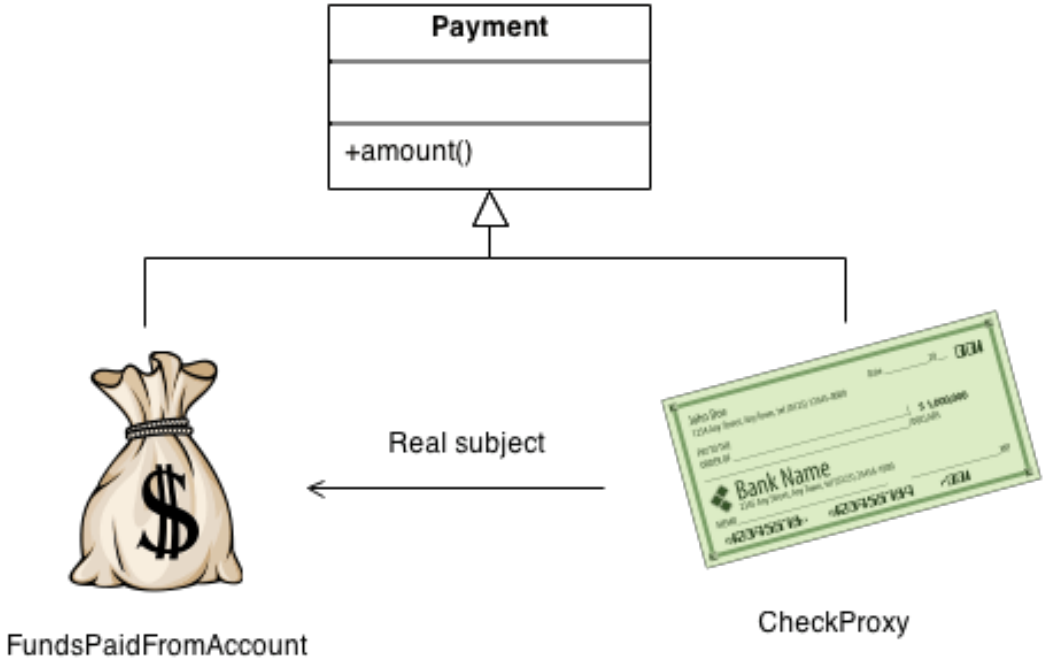
- A remote proxy provides a local representative for an object in a different address space.
- A virtual proxy creates expensive objects on demand.
- A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

Design participants

- **Subject –** is an interface which expose the functionality available to be used by the clients.
- **Real Subject –** is a class implementing Subject and it is concrete implementation which needs to be hidden behind a proxy.
- **Proxy –** hides the real object by extending it and clients communicate to real object via this proxy object. Usually frameworks create this proxy object when client request for real object.

Example

A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



Real-world example

- In corporate networks, internet access is guarded behind a network proxy. All network requests goes through proxy which first check the requests for allowed websites and posted data to network. If request looks suspicious, proxy block the request – else request pass through.

When to use proxy design pattern- Applicability

The Proxy Pattern can be used in scenarios like, when data can be cached to improve responsiveness, the Real Subject is in a remote location and requires wrapping and unwrapping of requests and making remote calls. The Real Subject should be protected from unauthorized access, or they are expensive to create. A few of the different types of Proxy Pattern implementation are discussed below:

- **Cache Proxy(**Caching request results)

  The Cache Proxy **improves the responsiveness by caching relevant data** (that does not change frequently or is expensive to create) at the Proxy object level, when a client sends a request, the Proxy checks if the requested data is present with it if the data is found, it is returned to the client without sending the request to the Real Object.

- **Remote Proxy(**Local execution of a remote service)

  The Remote Proxy provides a **proxy object at a local level which represents an object present at another location**.

When to use proxy design pattern

- **Protection Proxy(**Access control)

  As its name suggests, the Protection Proxy is used to provide some **security for Real Subject at the proxy level**

- **Virtual Proxy(**Lazy initialization)

  Virtual Proxy is used in a scenario when the **Real Object is a complex object or expensive to create**. This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.

Advantages of Proxy Design Pattern

- A few of the advantages of the Proxy Design Pattern are as follows:

  The proxy works even if the service object isn't ready or is not available.

  Open/Closed Principle - new proxies can be introduced without changing the service or clients.

Disadvantages of Proxy Design Pattern

- A few of the disadvantages of the Proxy Design Pattern are as follows:

The Proxy Design Pattern introduces an additional layer between the client and the Real Subject, this contributes to the code complexity.

The response from the service might get delayed.

Additional request forwarding is introduced between the client and the Real Subject.

Usage examples:

1.Image viewer program that lists and displays high resolution photos. The program has to show a list of all photos however it does not need to display the actual photo until the user selects an image item from a list.

2.The same for document editor that can embed graphical objects in a document. It isn't necessary to load all pictures when the document is opened, because not all of these objects will be visible at the same time.

3.The protective proxy acts as an authorization layer to verify if the actual user has access to appropriate content. An example can be thought about the proxy server which provides restrictive internet access in office. Only the websites and contents which are valid will be allowed and the remaining ones will be blocked.

4.Maybe used also for adding a thread-safe feature to an existing class without changing the existing class's code.

Example

Let’s take a scenario where the real image contains a huge size data which clients needs to access. To save our resources and memory the implementation will be as below:

– Create an interface which will be accessed by the client. All its methods will be implemented by the ProxyImage class and RealImage class.

– RealImage runs on the different system and contains the image information is accessed from the database.

– The ProxyImage which is running on a different system can represent the RealImage in the new system. Using the proxy we can avoid multiple loading of the image.

Example

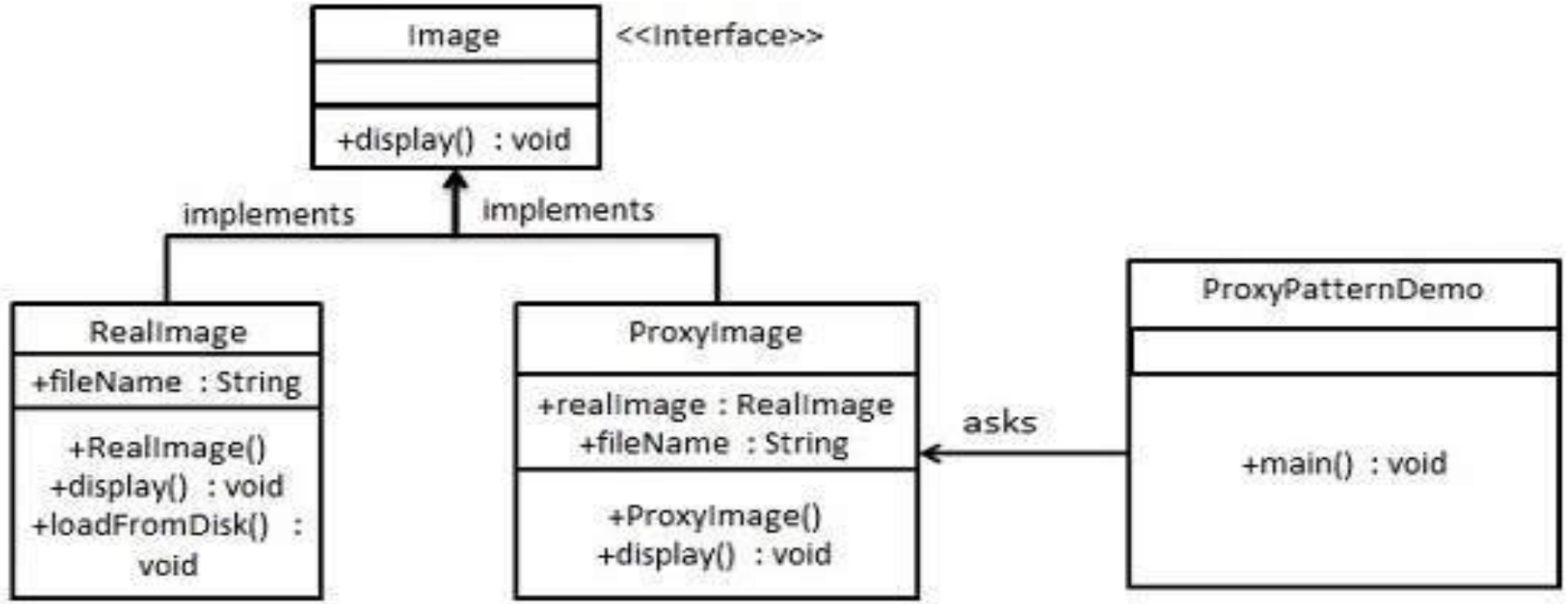
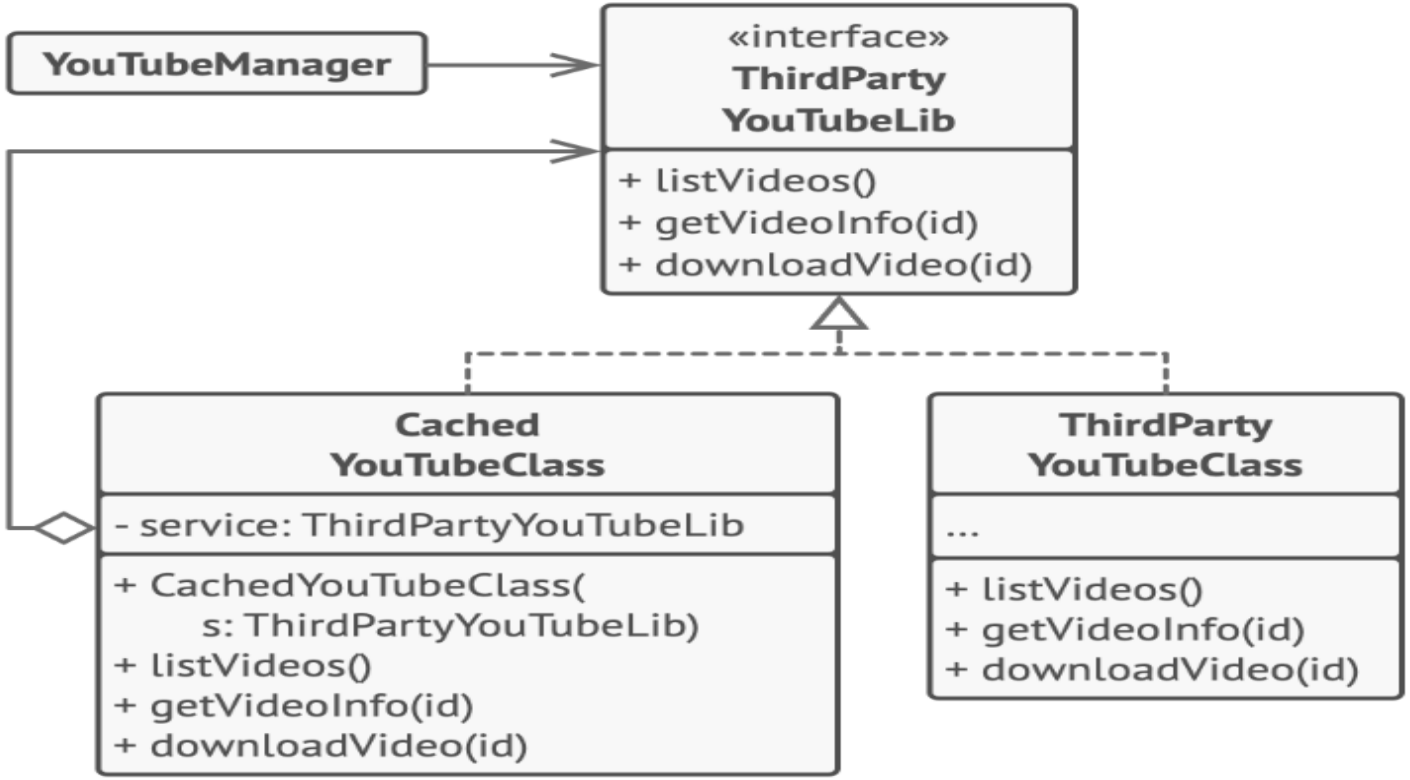


Image
<<Interface>>
+display() : void
implements
implements
RealImage
+fileName : String
+RealImage()
+display() : void
+loadFromDisk() : void
ProxyImage
+realImage : RealImage
+fileName : String
+ProxyImage()
+display() : void
asks
ProxyPatternDemo
+main() : void

Example

This example illustrates how the **Proxy** pattern can help to introduce lazy initialization and caching to a 3rd-party YouTube integration library.



Caching results of a service with a proxy.

Example

The library provides us with the video downloading class. However, it's very inefficient. If the client application requests the same video multiple times, the library just downloads it over and over, instead of caching and reusing the first downloaded file.

The proxy class implements the same interface as the original downloader and delegates it all the work. However, it keeps track of the downloaded files and returns the cached result when the app requests the same video multiple times.

THANK YOU

Prof. Priya Badarinath

Department of Computer Science and Engineering

Object Oriented Analysis and Design with Java

UE21CS352B

Prof. Priya Badarinath
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352B : Object Oriented Analysis and Design with Java

Unit 4
Structural Patterns - Flyweight

Department of Computer Science and Engineering

Agenda

- ❑ Flyweight-definition
- ✔ Intent
- ✔ Motivation
- ✔ Structure
- ✔ Applicability
- ✔ Consequence
- ✔ Implementation
- ✔ Issues

Flyweight design pattern - Introduction

- The flyweight pattern is for sharing objects, where each instance does not contain its own state but stores it externally.
- This allows efficient sharing of objects to save space when there are many instances but a few different types.

Flyweight design pattern - Intent

- Use sharing to support large number of fine-grained objects efficiently.
- Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Flyweight Pattern- Motivation

- An application uses many objects.
- A flyweight is a shared object that can be used in multiple contexts simultaneously.
- It acts as an individual object in each context.
- The key concept is the distinction between intrinsic and extrinsic state.

Problem

Placing the same information in many different places-

1. Mistakes can be made in copying the information.
2. If the data changes, all occurrences of the information must be located and changed. This makes maintenance of the document difficult.
3. Documents can become quite large if the same information is repeated over and over again.

Issues –

4. RAM overhead associated with each Instance.
5. CPU overhead associated with Memory management.

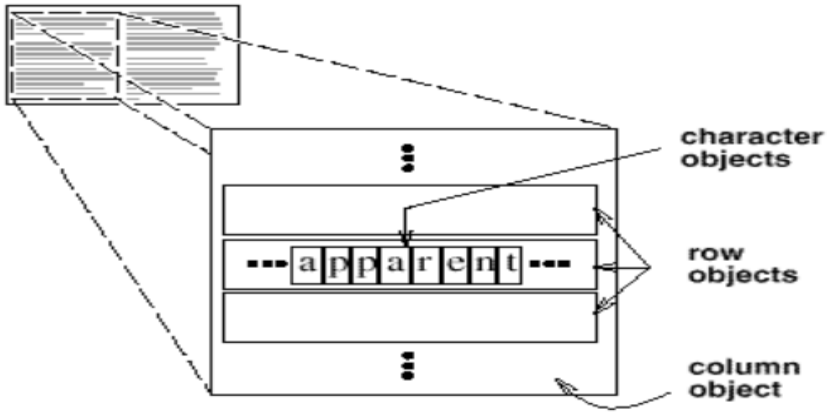
Solution

Create Flyweight Objects for Intrinsic state

- **Intrinsic State** – Information independent of the object's context, sharable(e.g state might include name, postal abbreviation, time zone etc.). This is included in the flyweight and instead of storing the same information n times for n objects, it is stored only once.
- **Extrinsic state** – Information dependent of the object's context, unshareable, stateless, having no stored values, but values that can be calculated on the spot(e.g. state might need access to region). This is excluded from the Flyweight.

Example: OO Document Editor

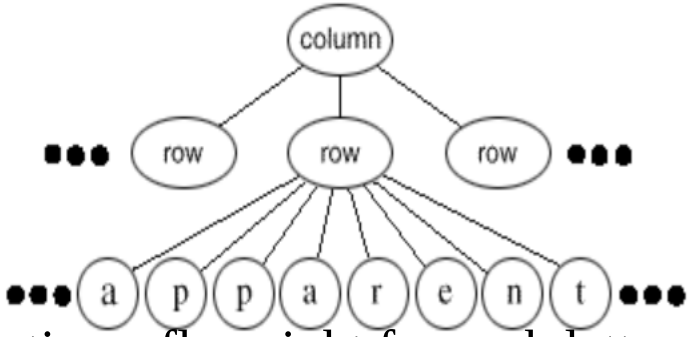
- Using an object for each character in the document:
- Promotes flexibility at the finest levels in the application.
- Character and embedded elements can be treated uniformly with respect to how they are drawn and formatted.
- The application could be extended to support new character sets without disturbing other functionalities.
- The application's object structure could mimic the document's physical structure.



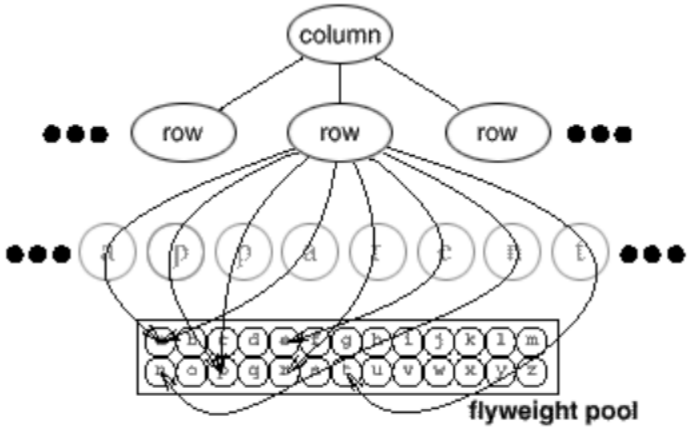
Object Oriented Analysis and Design with Java

Example: OO Document Editor

Logically



physically



Creating a flyweight for each letter of the alphabet:

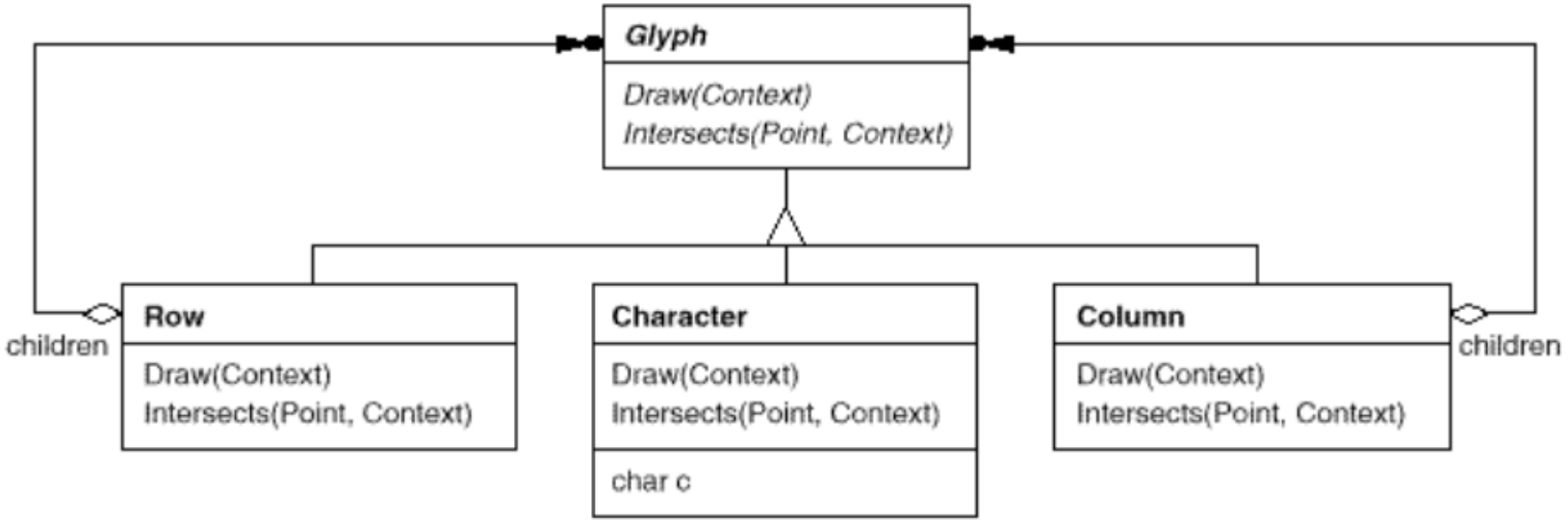
Intrinsic state: A character code

Extrinsic state: Coordinate position in the document

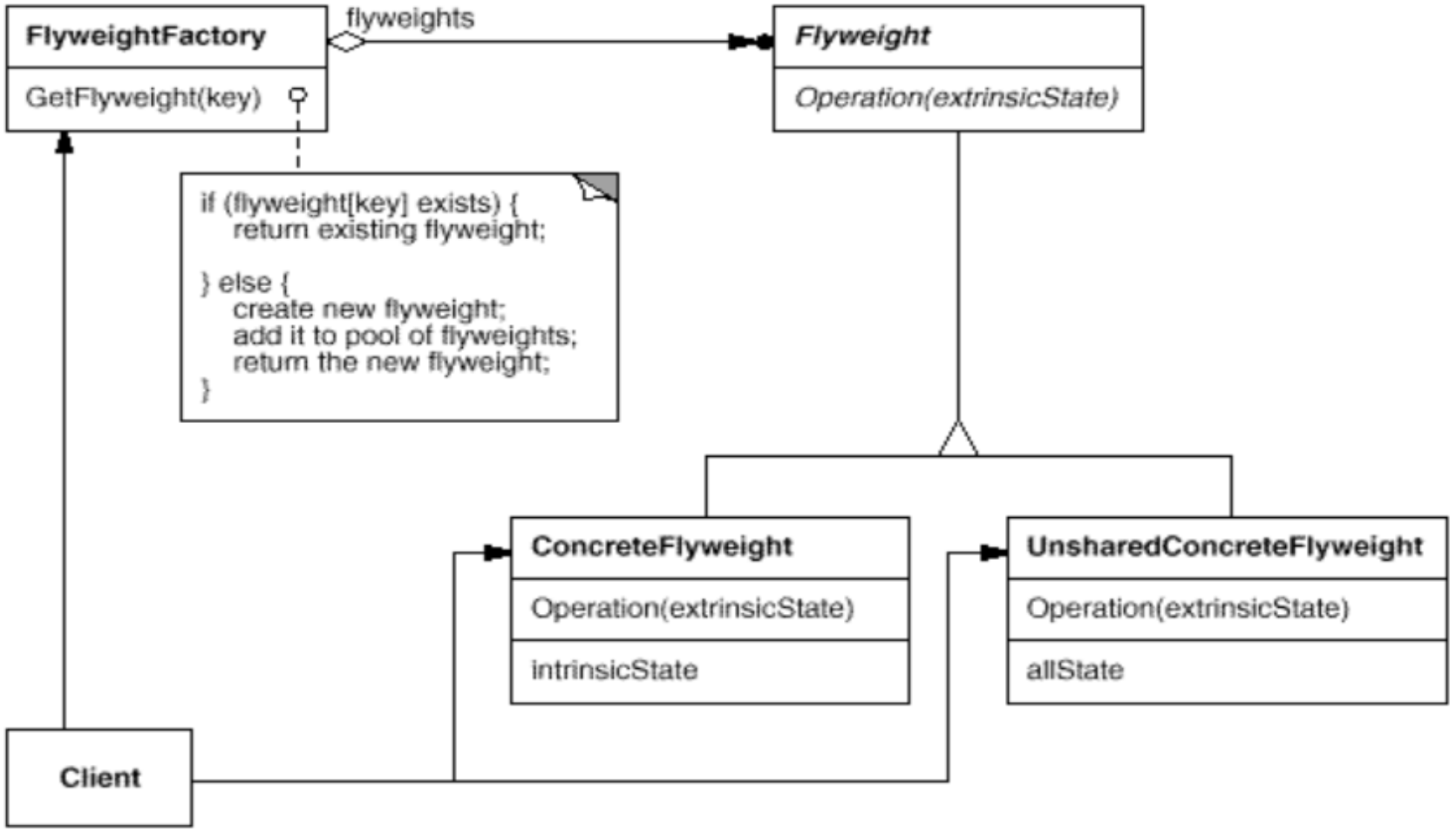
Typographic style(font, color)

Is determined from the text layout algorithms and formatting commands in effect wherever the character appears.

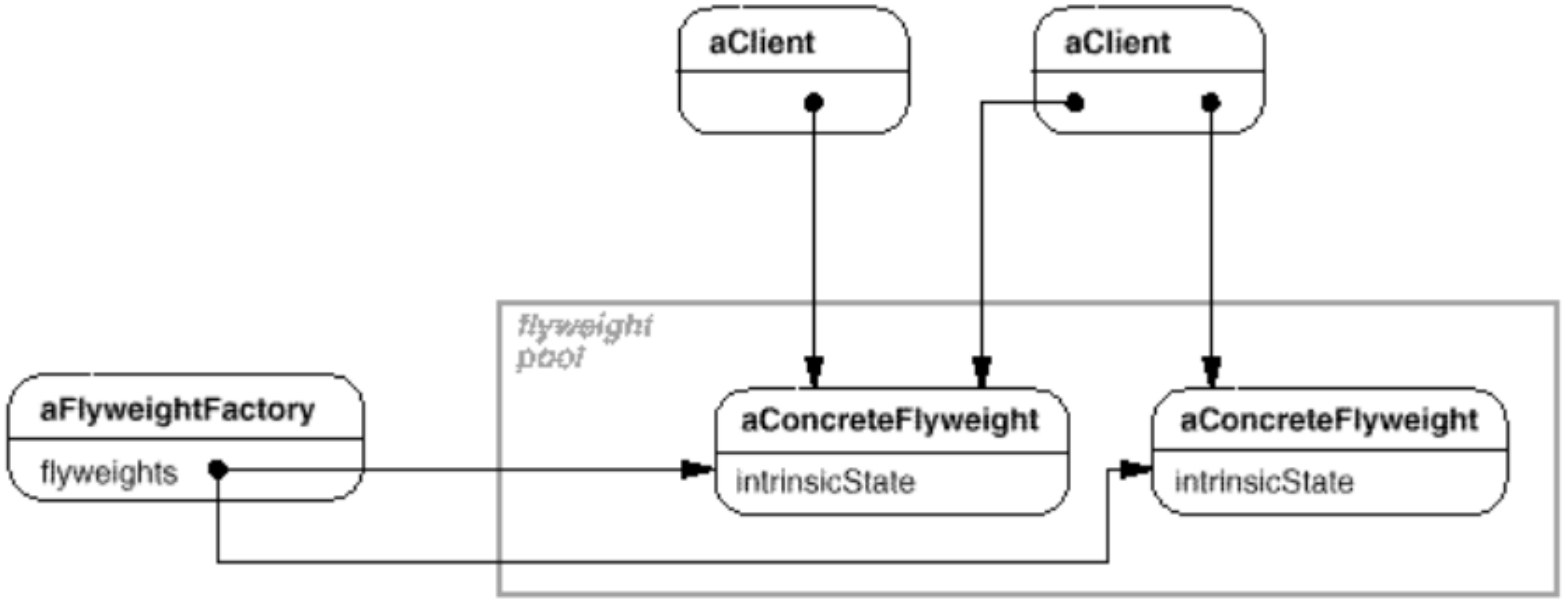
Class Structure



Structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Sharing of flyweights



Flyweight Participants

- **Flyweight**
  - declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight** (Character)
  - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.
- **UnsharedConcreteFlyweight** (Row, Column)
  - not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
- **FlyweightFactory**
  - creates and manages flyweight objects.
  - ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.
- **Client**
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s).

Applicability

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Consequences

- Disadvantage: Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state.
- Advantage: Storage savings are a function of several factors:
  - The reduction in the total number of instances that comes from sharing.
  - The amount of intrinsic state per object
  - Whether extrinsic state is computed or stored.

Implementation

- Removing extrinsic state - The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects. Removing extrinsic state won't help reduce storage costs if there are as many different kinds of extrinsic state as there are objects before sharing.

- Managing shared objects - Because objects are shared, clients shouldn't instantiate them directly. FlyweightFactory lets clients locate a particular flyweight. FlyweightFactory objects often use an associative store to let clients look up flyweights of interest.

Examples:

•Suppose we have a **pen** which can exist with/without **refill**. A refill can be of any color thus a pen can be used to create drawings having N number of colors.
Here Pen can be flyweight object with refill as extrinsic attribute. All other attributes such as pen body, pointer etc. can be intrinsic attributes which will be common to all pens. A pen will be distinguished by its refill color only, nothing else.
All application modules which need to access a red pen – can use the same instance of red pen (shared object). Only when a different color pen is needed, application module will ask for another pen from flyweight factory.
•In programming, we can see **java.lang.String** constants as flyweight objects. All strings are stored in string pool and if we need a string with certain content then runtime return the reference to already existing string constant from the pool – if available.
•In browsers, we

• use an image in multiple places in a webpage. Browsers will load the image only one time, and for other times browsers will reuse the image from cache. Now image is same but used in multiple places. It's URL is intrinsic attribute because it's fixed and shareable. Images position coordinates, height and width are extrinsic attributes which vary according to place (context) where they have to be rendered.

References

- https://www.cs.unc.edu/~stotts/GOF/hires/pat4ffso.htm
- https://refactoring.guru/design-patterns/flyweight
- https://howtodoinjava.com/design-patterns/structural/flyweight-design-pattern/
- Design Patterns: Elements of Reusable Object-Oriented SoftwareErich Gamma, Ralph Johnson, Richard Helm · 1995

THANK YOU

Prof. Priya Badarinath

Department of Computer Science and Engineering