M2006C3MII

6:21 PM

UNIT - 4

Syllabus:

Behavioral Patterns: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor, Discussion of Behavioral Patterns

Behavioral Patterns:

Chain of Responsibility

Intent

In writing an application of any kind, it often happens that the event generated by one object needs to be handled by another one. And, to make our work even harder, we also

Type here to search 28°C ENG 18:19 03-08-2023

M2006C3MII

6:22 PM

UNIT - 4

Syllabus:

Behavioral Patterns: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor, Discussion of Behavioral Patterns

Behavioral Patterns:

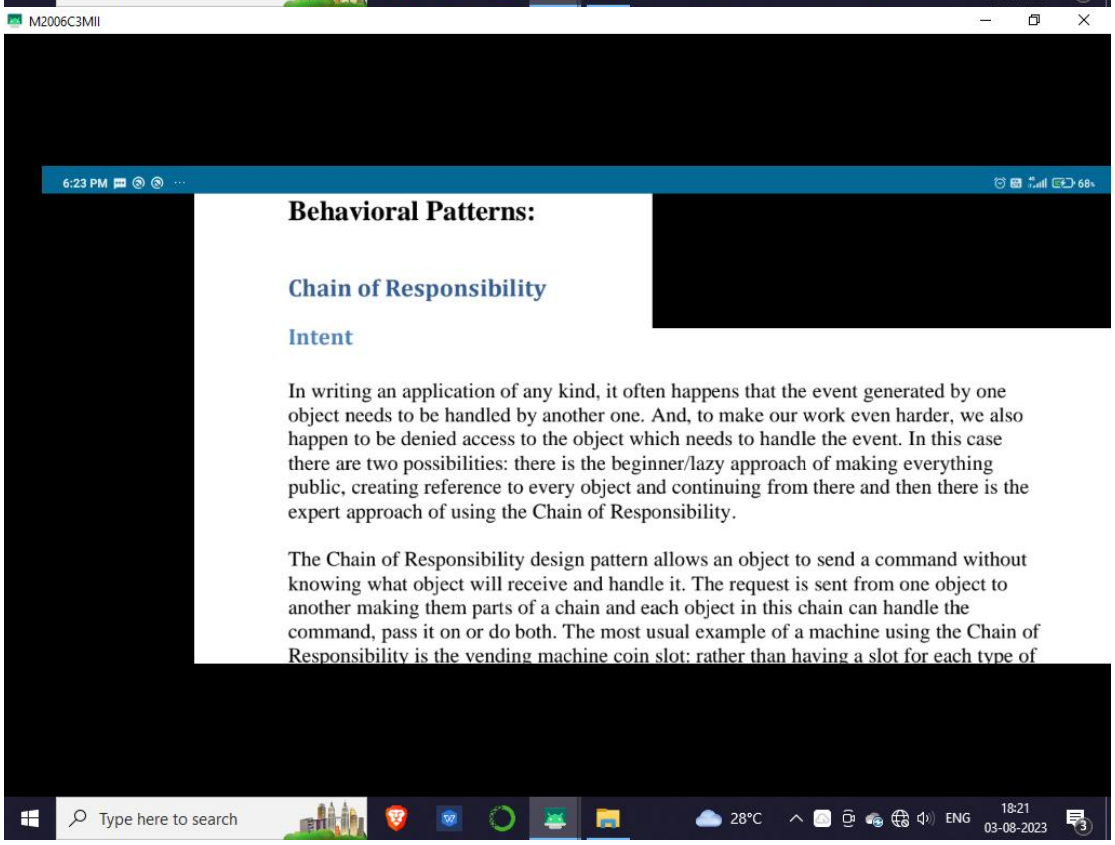
Chain of Responsibility

Intent

In writing an application of any kind, it often happens that the event generated by one object needs to be handled by another one. And, to make our work even harder, we also

Type here to search 28°C ENG 18:20 03-08-2023


M2006C3MII
6:23 PM
Behavioral Patterns:
Chain of Responsibility
Intent
In writing an application of any kind, it often happens that the event generated by one object needs to be handled by another one. And, to make our work even harder, we also happen to be denied access to the object which needs to handle the event. In this case there are two possibilities: there is the beginner/lazy approach of making everything public, creating reference to every object and continuing from there and then there is the expert approach of using the Chain of Responsibility.
The Chain of Responsibility design pattern allows an object to send a command without knowing what object will receive and handle it. The request is sent from one object to another making them parts of a chain and each object in this chain can handle the command, pass it on or do both. The most usual example of a machine using the Chain of Responsibility is the vending machine coin slot: rather than having a slot for each type of
Type here to search
28°C
ENG
18:21
03-08-2023


M2006C3MII
6:23 PM
Behavioral Patterns:
Chain of Responsibility
Intent
In writing an application of any kind, it often happens that the event generated by one object needs to be handled by another one. And, to make our work even harder, we also happen to be denied access to the object which needs to handle the event. In this case there are two possibilities: there is the beginner/lazy approach of making everything public, creating reference to every object and continuing from there and then there is the expert approach of using the Chain of Responsibility.
The Chain of Responsibility design pattern allows an object to send a command without knowing what object will receive and handle it. The request is sent from one object to another making them parts of a chain and each object in this chain can handle the command, pass it on or do both. The most usual example of a machine using the Chain of Responsibility is the vending machine coin slot: rather than having a slot for each type of
Type here to search
28°C
ENG
18:21
03-08-2023

M2006C3MII

6:23 PM

The Chain of Responsibility design pattern allows an object to send a command without knowing what object will receive and handle it. The request is sent from one object to another making them parts of a chain and each object in this chain can handle the command, pass it on or do both. The most usual example of a machine using the Chain of Responsibility is the vending machine coin slot: rather than having a slot for each type of coin, the machine has only one slot for all of them. The dropped coin is routed to the appropriate storage place that is determined by the receiver of the command.

Intent:

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

Implementation

The UML diagram of classes below will help us understand better the way the Chain works.

Type here to search 28°C ENG 18:21 03-08-2023

M2006C3MII

6:23 PM

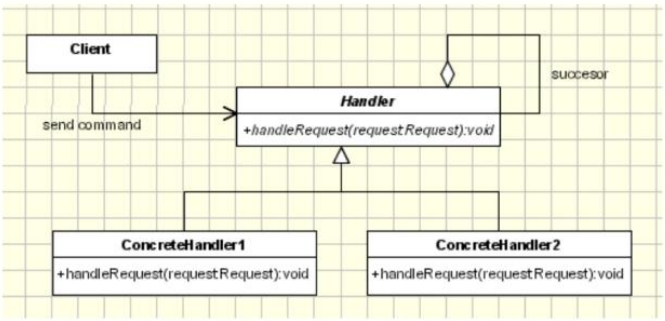
Implementation

The UML diagram of classes below will help us understand better the way the Chain works.

Type here to search 28°C ENG 18:22 03-08-2023

M2006C3MII

6:23 PM



In the diagram above some explanations are needed on what is the role of every class:

- **Handler** - defines an interface for handling requests
  - **RequestHandler** - handles the requests it is responsible for

Type here to search 28°C ENG 18:22 03-08-2023

M2006C3MII

6:24 PM

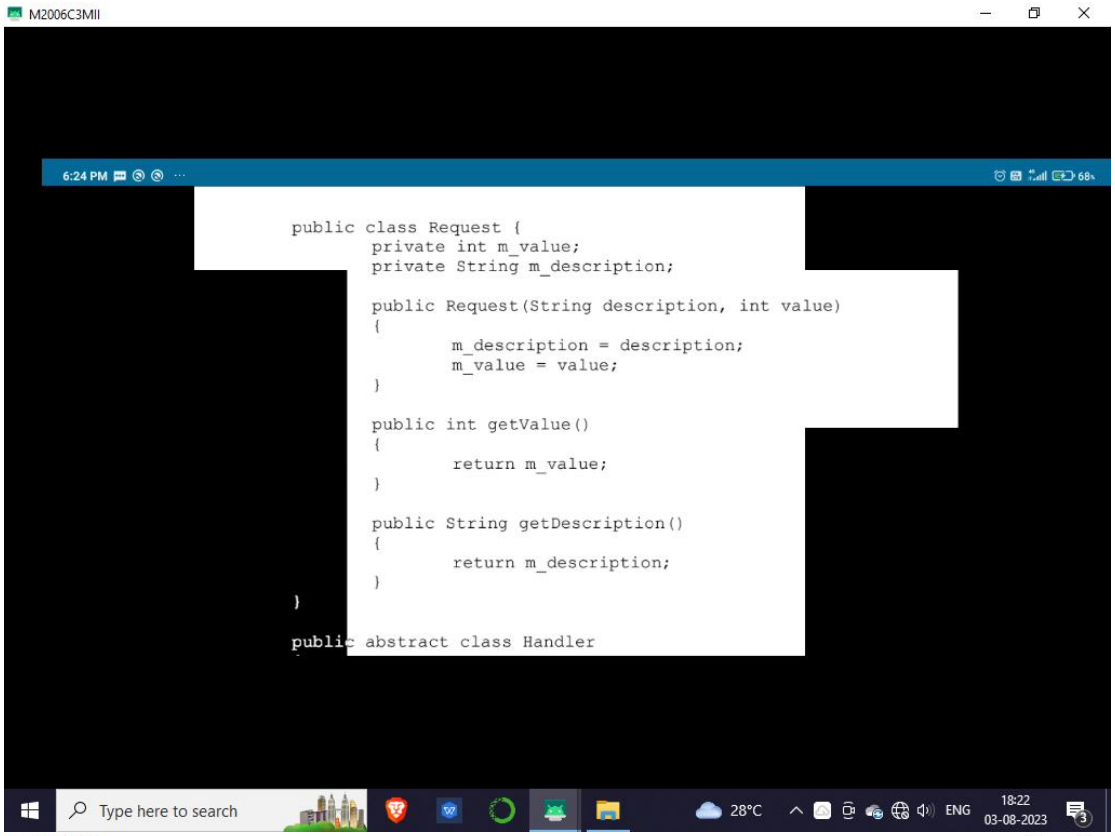
In the diagram above some explanations are needed on what is the role of every class:

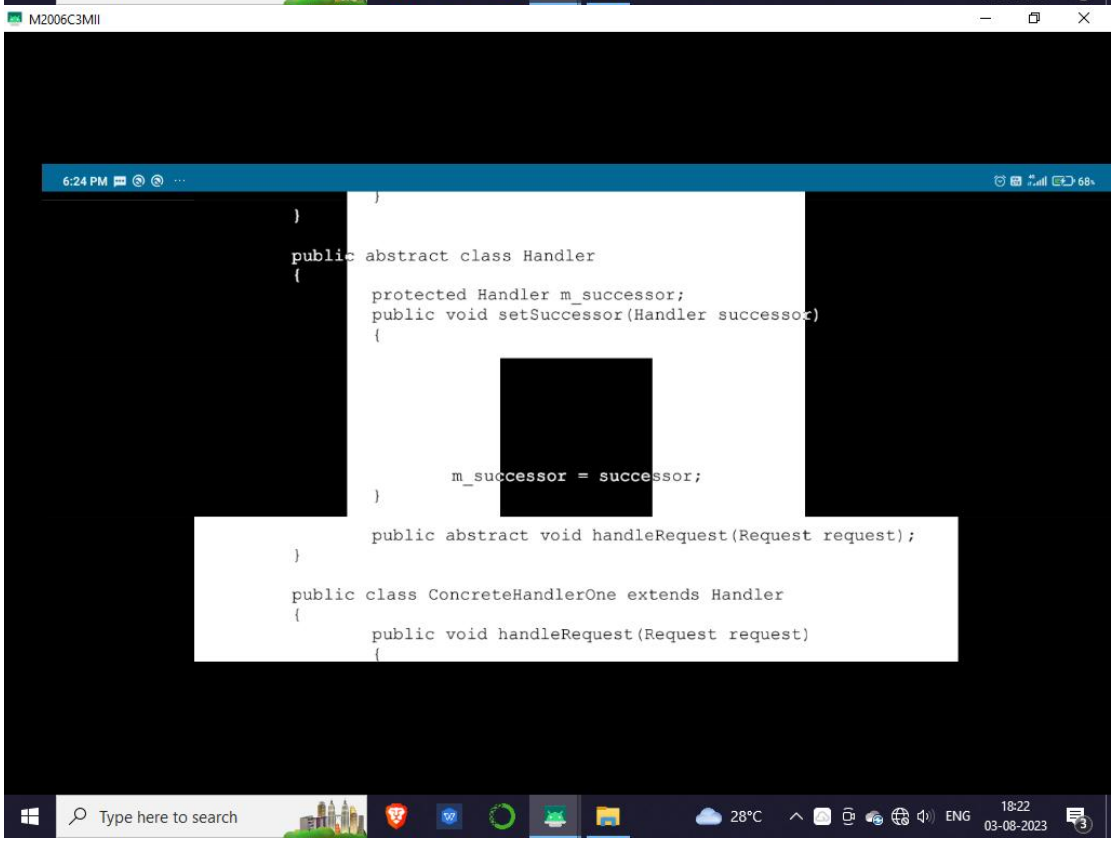
- **Handler** - defines an interface for handling requests
  - **RequestHandler** - handles the requests it is responsible for
  - If it can handle the request it does so, otherwise it sends the request to its successor
- **Client** - sends commands to the first object in the chain that may handle the command

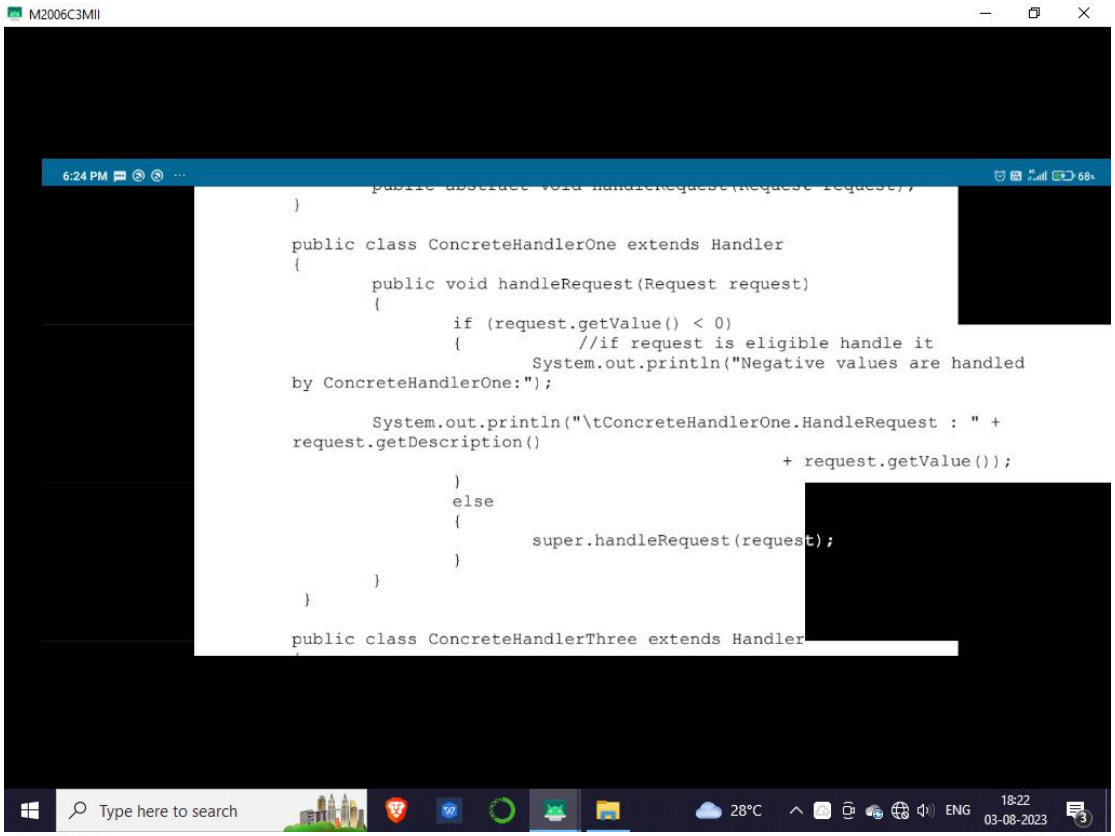
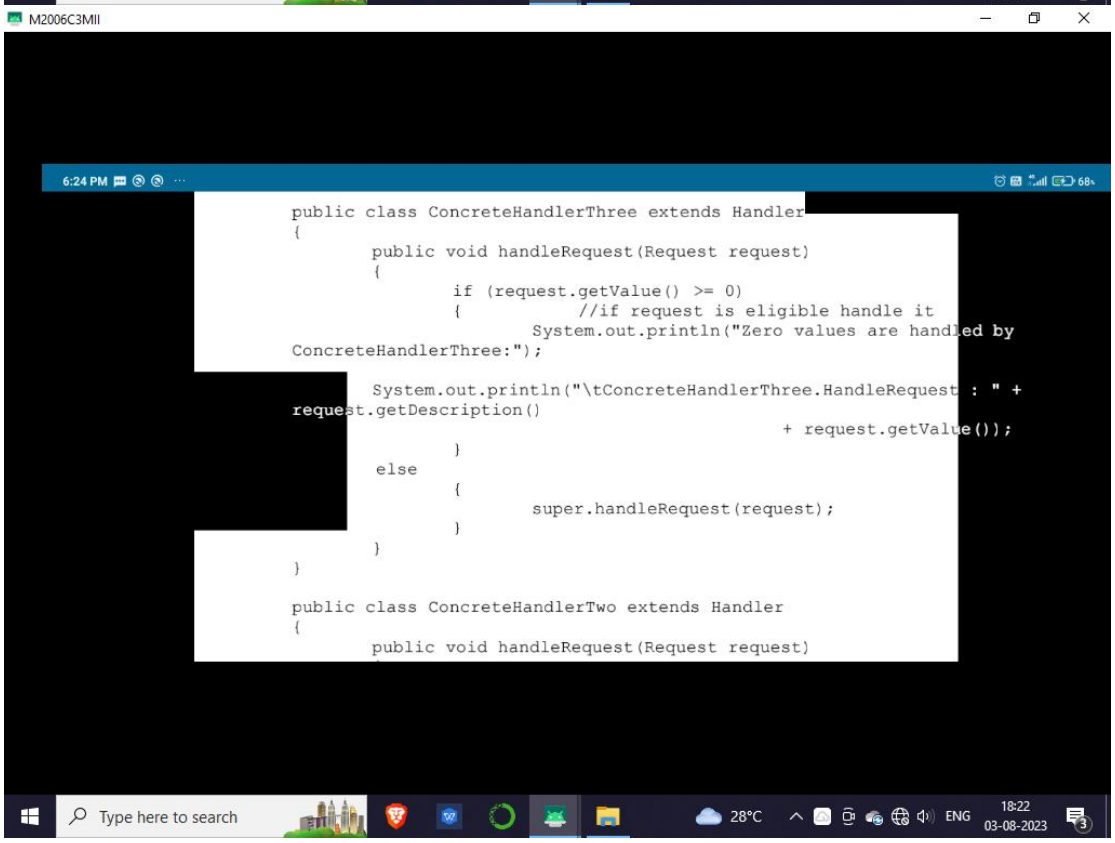
Here is how sending a request works in the application using the Chain of Responsibility: the Client in need of a request to be handled sends it to the chain of handlers, which are classes that extend the Handler class. Each of the handlers in the chain takes its turn at trying to handle the request it receives from the client. If ConcreteHandler_i can handle it, then the request is handled, if not it is sent to the handler ConcreteHandler_i+1, the next one in the chain.

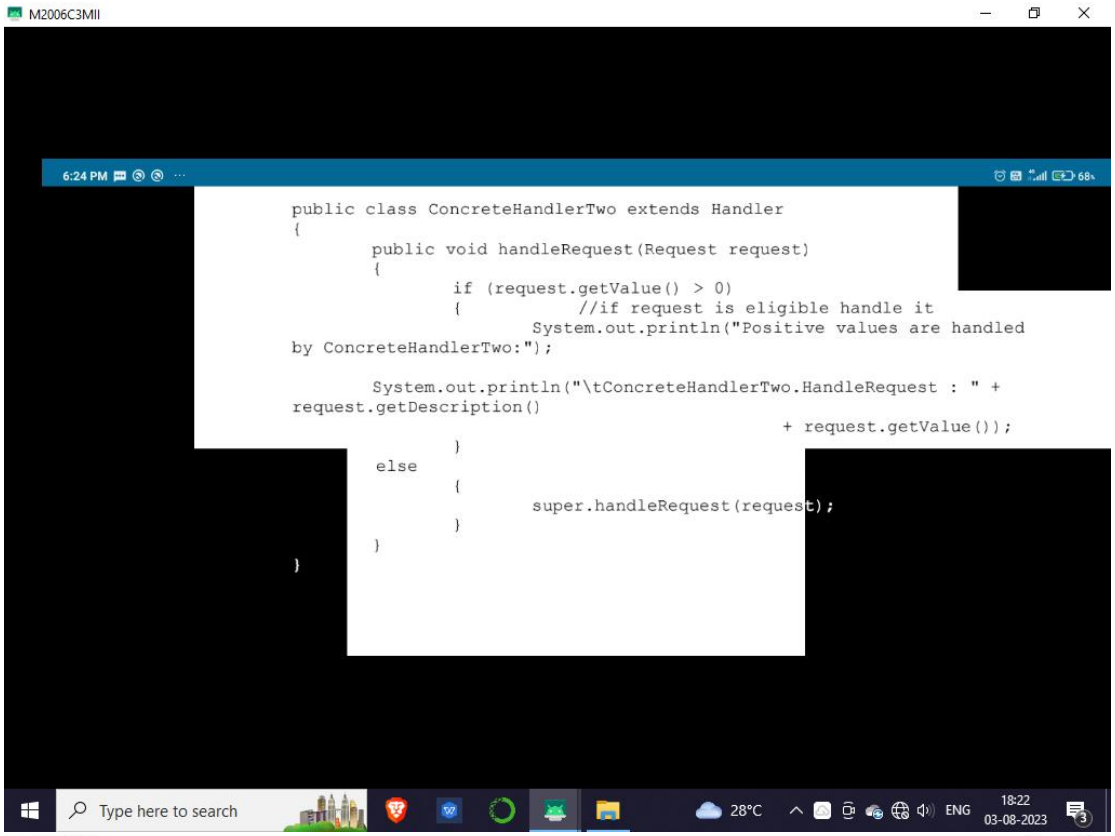
The classic example of the Chain of Responsibility's implementation is presented for us below:

Type here to search 28°C ENG 18:22 03-08-2023

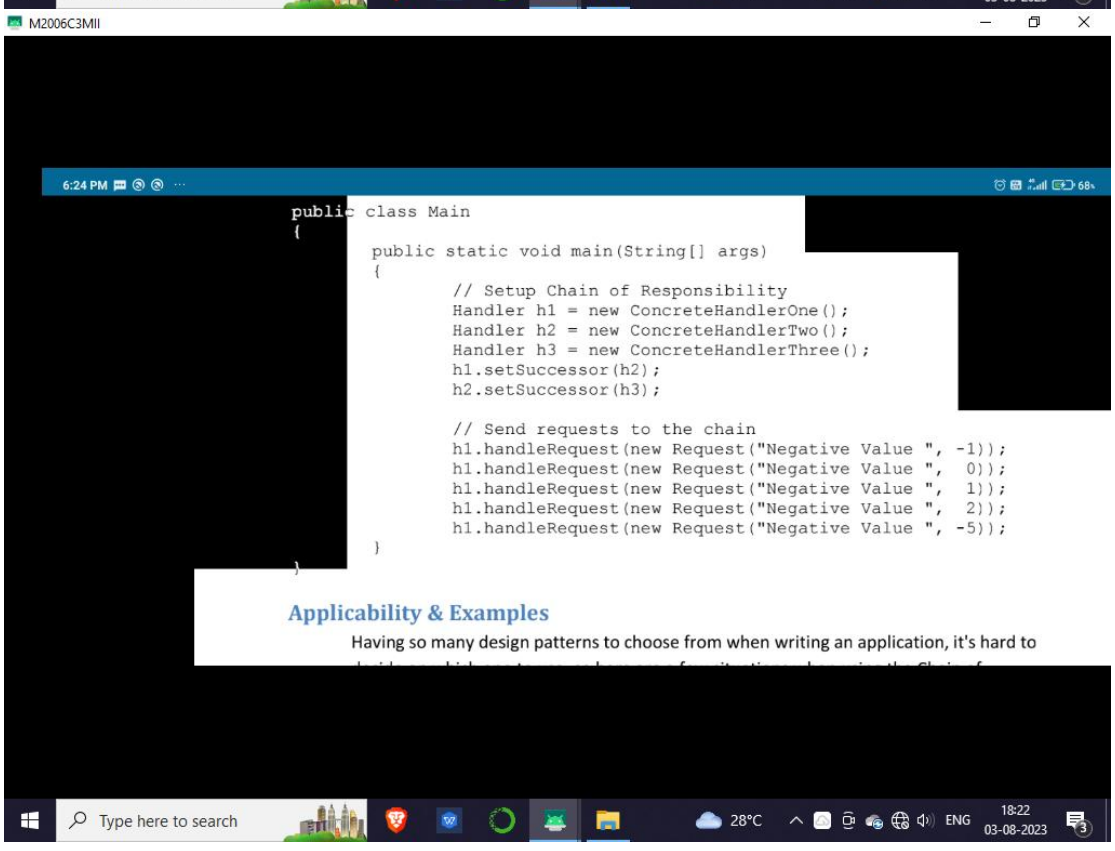

M2006C3MII
6:24 PM
public class Request {
private int m_value;
private String m_description;
public Request(String description, int value)
{
m_description = description;
m_value = value;
}
public int getValue()
{
return m_value;
}
public String getDescription()
{
return m_description;
}
}
public abstract class Handler
Type here to search
28°C
ENG
18:22
03-08-2023


M2006C3MII
6:24 PM
}
public abstract class Handler
{
protected Handler m_successor;
public void setSuccessor(Handler successor)
{
m_successor = successor;
}
public abstract void handleRequest(Request request);
}
public class ConcreteHandlerOne extends Handler
{
public void handleRequest(Request request)
{
Type here to search
28°C
ENG
18:22
03-08-2023


M2006C3MII
6:24 PM
}
public class ConcreteHandlerOne extends Handler
{
public void handleRequest(Request request)
{
if (request.getValue() < 0)
{ //if request is eligible handle it
System.out.println("Negative values are handled
by ConcreteHandlerOne:");
System.out.println("\tConcreteHandlerOne.HandleRequest : " +
request.getDescription()
+ request.getValue());
}
else
{
super.handleRequest(request);
}
}
}
public class ConcreteHandlerThree extends Handler
Type here to search
28°C
ENG
18:22
03-08-2023

M2006C3MII
6:24 PM
public class ConcreteHandlerThree extends Handler
{
public void handleRequest(Request request)
{
if (request.getValue() >= 0)
{ //if request is eligible handle it
System.out.println("Zero values are handled by
ConcreteHandlerThree:");
System.out.println("\tConcreteHandlerThree.HandleRequest : " +
request.getDescription()
+ request.getValue());
}
else
{
super.handleRequest(request);
}
}
}
public class ConcreteHandlerTwo extends Handler
{
public void handleRequest(Request request)
Type here to search
28°C
ENG
18:22
03-08-2023



```
public class ConcreteHandlerTwo extends Handler
{
        public void handleRequest(Request request)
        {
                if (request.getValue() > 0)
                {           //if request is eligible handle it
                        System.out.println("Positive values are handled
by ConcreteHandlerTwo:");

        System.out.println("\tConcreteHandlerTwo.HandleRequest : " +
request.getDescription()
                                                + request.getValue());
                }
         else
                {
                        super.handleRequest(request);
                }
        }
}
```



```
public class Main
{
        public static void main(String[] args)
        {
                // Setup Chain of Responsibility
                Handler h1 = new ConcreteHandlerOne();
                Handler h2 = new ConcreteHandlerTwo();
                Handler h3 = new ConcreteHandlerThree();
                h1.setSuccessor(h2);
                h2.setSuccessor(h3);

                // Send requests to the chain
                h1.handleRequest(new Request("Negative Value ", -1));
                h1.handleRequest(new Request("Negative Value ",  0));
                h1.handleRequest(new Request("Negative Value ",  1));
                h1.handleRequest(new Request("Negative Value ",  2));
                h1.handleRequest(new Request("Negative Value ", -5));
        }
}
```

Applicability & Examples

Having so many design patterns to choose from when writing an application, it's hard to

M2006C3MII

6:24 PM

Applicability & Examples

Having so many design patterns to choose from when writing an application, it's hard to decide on which one to use, so here are a few situations when using the Chain of Responsibility is more effective:

- More than one object can handle a command
- The handler is not known in advance
- The handler should be determined automatically
- It's wished that the request is addressed to a group of objects without explicitly specifying its receiver
- The group of objects that may handle the command must be specified in a dynamic way

Here are some real situations in which the Chain of Responsibility is used:

Example 1

In designing the software for a system that approves the purchasing requests.

In this case, the values of purchase are divided into categories, each having its own

Type here to search 28°C ENG 18:22 03-08-2023

M2006C3MII

6:24 PM

Example 1

In designing the software for a system that approves the purchasing requests.

In this case, the values of purchase are divided into categories, each having its own approval authority. The approval authority for a given value could change at any time and the system should be flexible enough to handle the situation.

The Client in the example above is the system in need of the answer to the approval. It sends a request about it to an purchase approval authority. Depending on the value of the purchase, this authority may approve the request or forward it to the next authority in the chain.

For example let's say a request is placed for the purchase of a new keyboard for an office. The value of the purchase is not that big, so the request is sent from the head of the office to the head of the department and then to the materials department where it stops, being handled locally. But if equipment for the whole department is needed then the request goes form the head of the department, to materials department, to the purchase office and even to the manager if the value is too big.

Type here to search 28°C ENG 18:22 03-08-2023

M2006C3MII

6:24 PM

Command Pattern

"An object that contains a symbol, name or key that represents a list of commands, actions or keystrokes". This is the definition of a macro, one that should be familiar to any computer user. From this idea the Command design pattern was given birth.

The Macro represents, at some extent, a command that is built from the reunion of a set of other commands, in a given order. Just as a macro, the Command design pattern encapsulates commands (method calls) in objects allowing us to issue requests without knowing the requested operation or the requesting object. Command design pattern provides the options to queue commands, undo/redo actions and other manipulations.

Intent

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

Type here to search 28°C ENG 18:22 03-08-2023

M2006C3MII

6:24 PM

Intent

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

Implementation

The idea and implementation of the Command design pattern is quite simple, as we will see in the diagram below, needing only few extra classes implemented.

Command pattern is a data driven design pattern and it falls under behavioral pattern category. A request is wrapped under the object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command.

In this demo example we've created an interface *Order* which is acting as a command. We've created a *StockInfo* class which acts as a request. We've concrete command classes *BuyStockInfo* and *SellStockInfo* implementing *Order*interface which will do actual command processing. A class *Broker* is created which acts as a invoker object. It can take order and place orders. *Broker* object uses command pattern to identify which object will execute which command based on type of command. *CommandPatternDemo*, our demo class will use *Broker* class to demonstrate command pattern.

Type here to search 28°C ENG 18:23 03-08-2023

M2006C3MII

6:24 PM

Implementation

The idea and implementation of the Command design pattern is quite simple, as we will see in the diagram below, needing only few extra classes implemented.

Command pattern is a data driven design pattern and it falls under behavioral pattern category. A request is wrapped under the object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command.

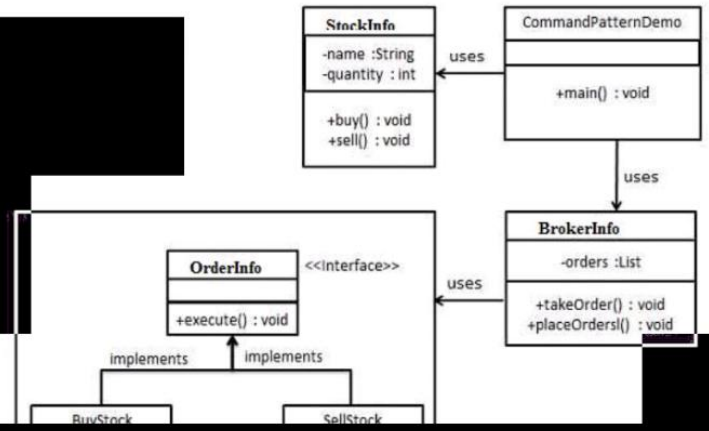
In this demo example we've created an interface *Order* which is acting as a command. We've created a *StockInfo* class which acts as a request. We've concrete command classes *BuyStockInfo* and *SellStockInfo* implementing *Order*interface which will do actual command processing. A class *Broker* is created which acts as a invoker object. It can take order and place orders. *Broker* object uses command pattern to identify which object will execute which command based on type of command. *CommandPatternDemo*, our demo class will use *Broker* class to demonstrate command pattern.

Type here to search 28°C ENG 18:23 03-08-2023

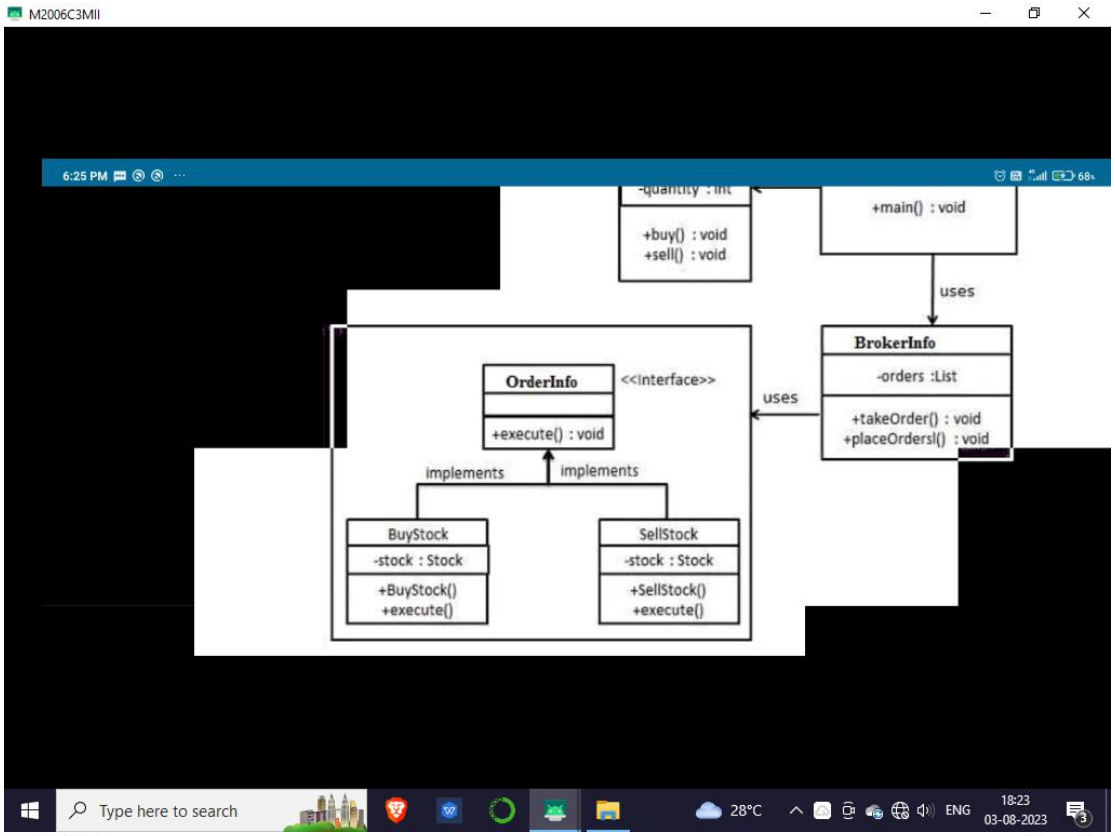
M2006C3MII

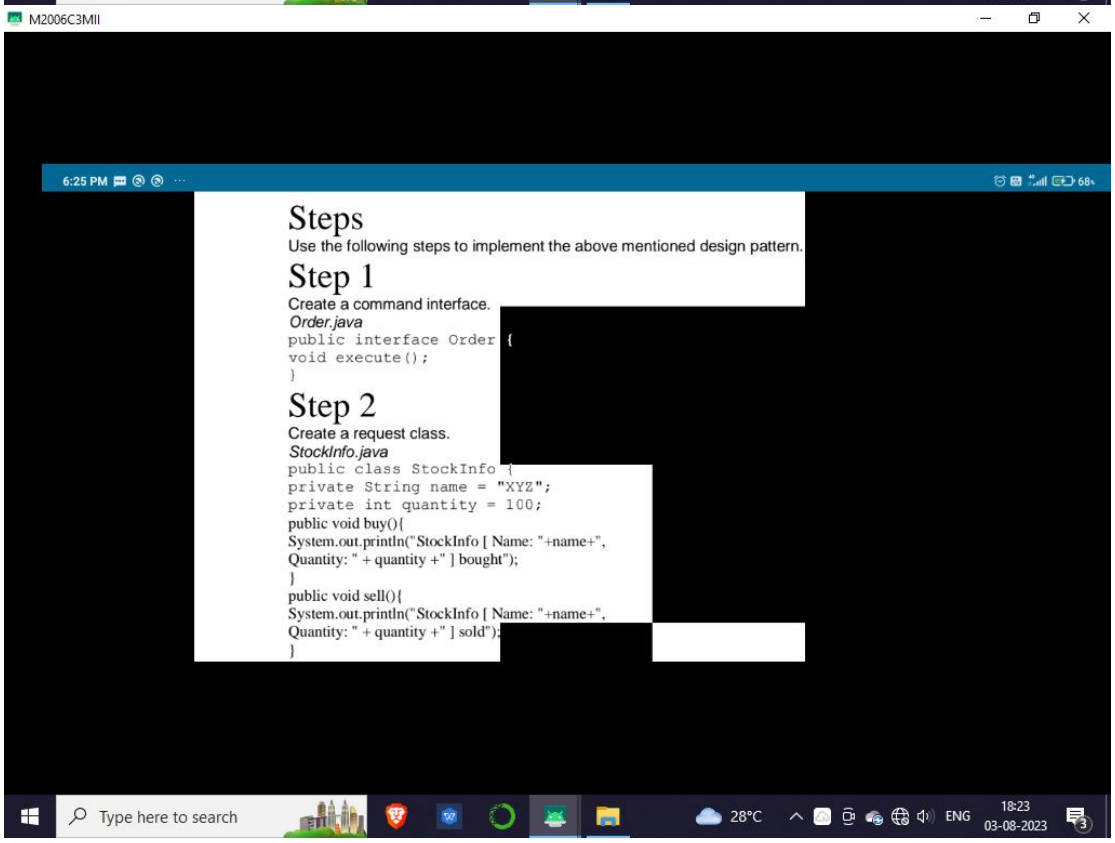
6:25 PM

Class Diagram:



Type here to search 28°C ENG 18:23 03-08-2023


M2006C3MII
6:25 PM
-quantity : int
+buy() : void
+sell() : void
+main() : void
uses
BrokerInfo
-orders :List
+takeOrder() : void
+placeOrdersl() : void
uses
OrderInfo
<<interface>>
+execute() : void
implements
implements
BuyStock
-stock : Stock
+BuyStock()
+execute()
SellStock
-stock : Stock
+SellStock()
+execute()
Type here to search
28°C
ENG
18:23
03-08-2023


M2006C3MII
6:25 PM
Steps
Use the following steps to implement the above mentioned design pattern.
Step 1
Create a command interface.
Order.java
public interface Order {
void execute();
}
Step 2
Create a request class.
StockInfo.java
public class StockInfo {
private String name = "XYZ";
private int quantity = 100;
public void buy(){
System.out.println("StockInfo [Name: "+name+",
Quantity: " + quantity +"] bought");
}
public void sell(){
System.out.println("StockInfo [Name: "+name+",
Quantity: " + quantity +"] sold");
}
Type here to search
28°C
ENG
18:23
03-08-2023

M2006C3MII

6:25 PM

Create a command interface.

Order.java

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

Step 2

Create a request class.

StockInfo.java

```
public class StockInfo {
private String name = "XYZ";
private int quantity = 100;
public void buy(){
System.out.println("StockInfo [ Name: "+name+",
Quantity: " + quantity +" ] bought");
}
public void sell(){
System.out.println("StockInfo [ Name: "+name+",
Quantity: " + quantity +" ] sold");
}
}
```

Step 3

Create concrete classes implementing the *Order* interface.

Type here to search

28°C ENG 18:23 03-08-2023

M2006C3MII

6:25 PM

BuyStock.java

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

SellStock.java

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

Step 4

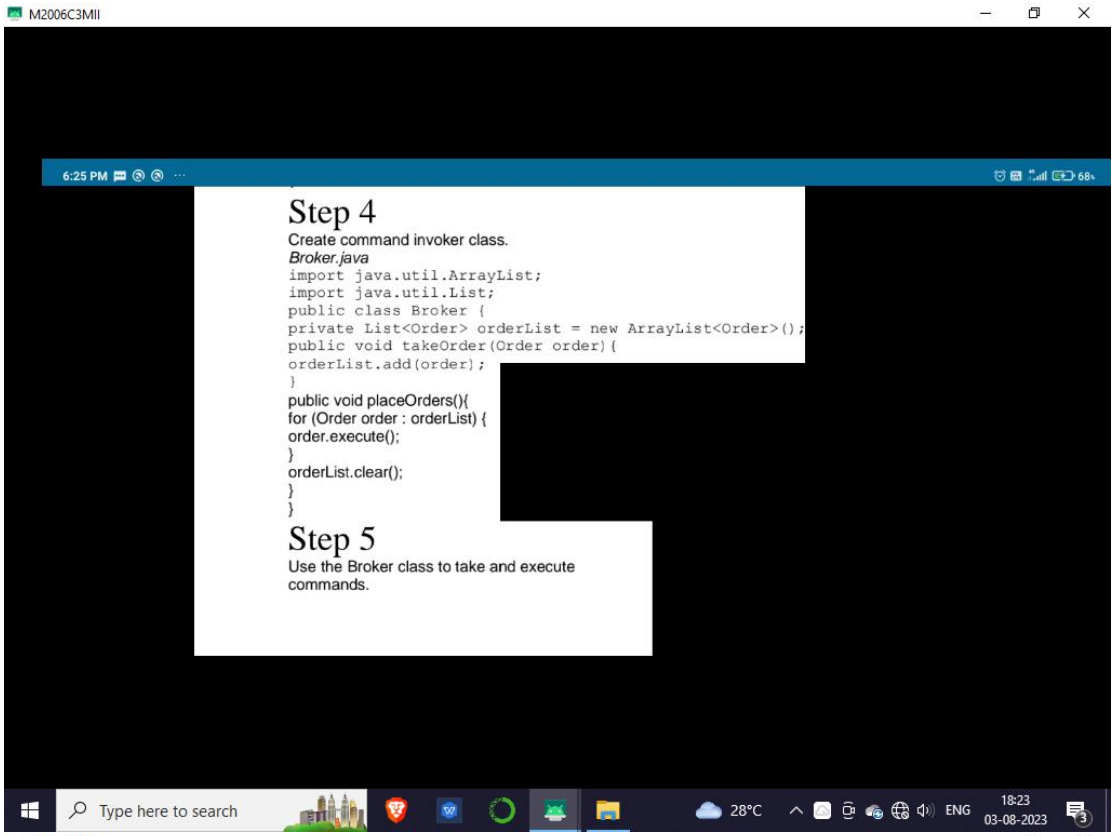
Create command invoker class.

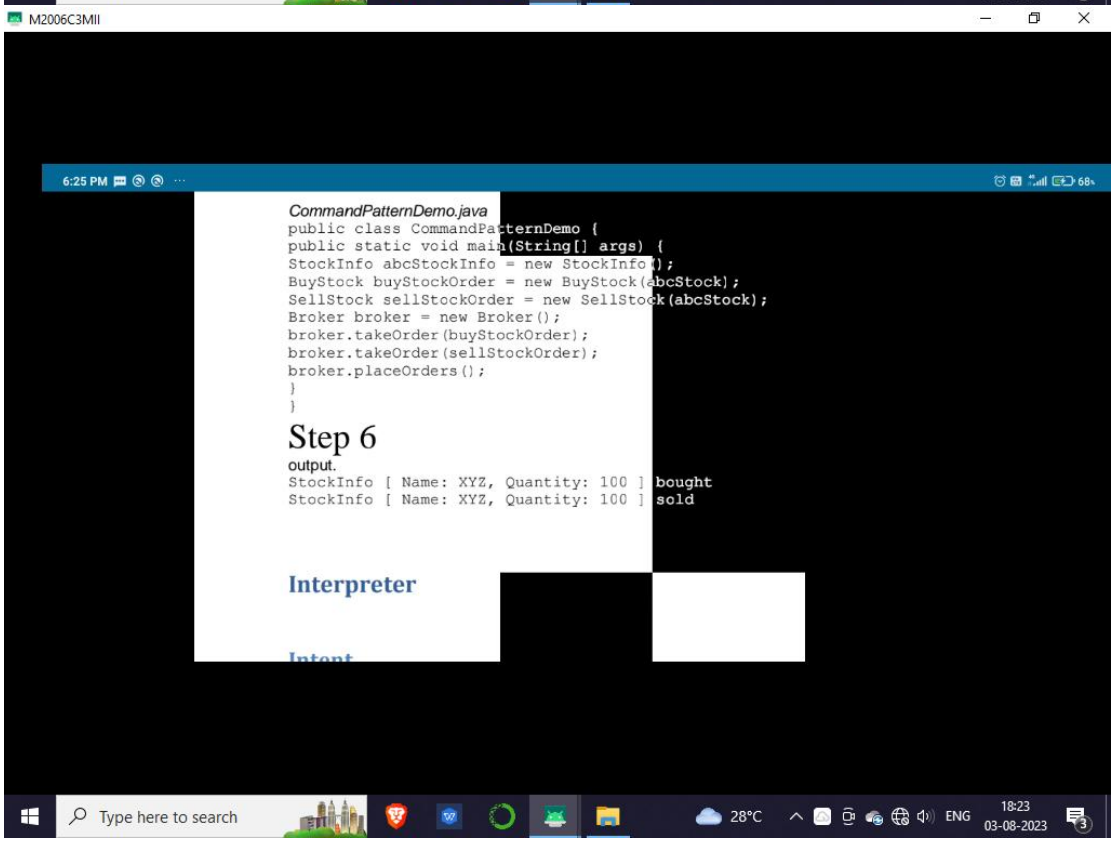
Broker.java

```
import java.util.ArrayList;
```

Type here to search

28°C ENG 18:23 03-08-2023


M2006C3MII
6:25 PM
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
Use the Broker class to take and execute commands.
Type here to search
28°C
ENG
18:23
03-08-2023


M2006C3MII
6:25 PM
CommandPatternDemo.java
public class CommandPatternDemo {
public static void main(String[] args) {
StockInfo abcStockInfo = new StockInfo();
BuyStock buyStockOrder = new BuyStock(abcStock);
SellStock sellStockOrder = new SellStock(abcStock);
Broker broker = new Broker();
broker.takeOrder(buyStockOrder);
broker.takeOrder(sellStockOrder);
broker.placeOrders();
}
}
Step 6
output.
StockInfo [Name: XYZ, Quantity: 100] bought
StockInfo [Name: XYZ, Quantity: 100] sold
Interpreter
Intent
Type here to search
28°C
ENG
18:23
03-08-2023

M2006C3MII

6:25 PM

Interpreter

Intent

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

Motivation

The Interpreter is one of the Design Patterns published in the GoF which is not really used. Usually the Interpreter Pattern is described in terms of formal grammars; like it was described in the original form in the GoF but the area where this design pattern can be applied can be extended.

Implementation

Type here to search 28°C ENG 18:23 03-08-2023

M2006C3MII

6:25 PM

Implementation

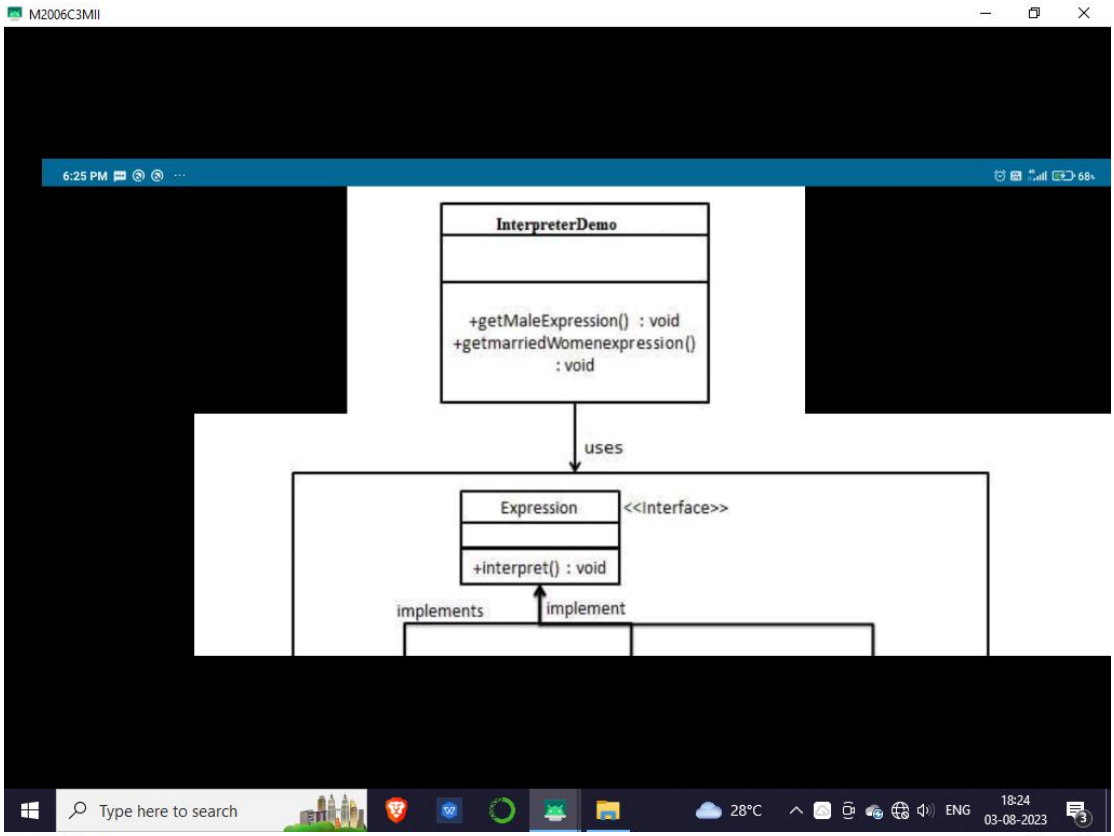
The implementation of the Interpreter pattern is just the use of the composite pattern applied to represent a grammar. The Interpreter defines the behavior while the composite defines only the structure.

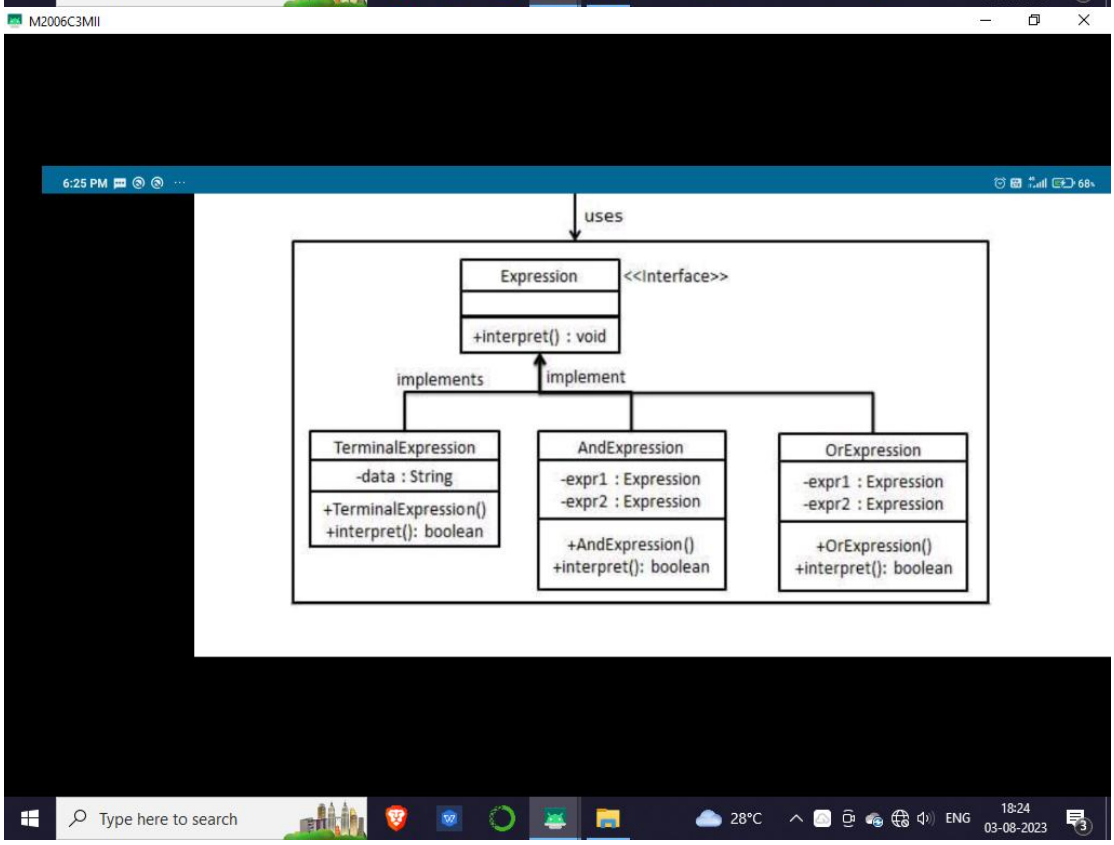
This type of pattern comes under behavioral patterns. This pattern involves implementing an expression interface which tells to interpret a particular context. This pattern is used in SQL parsing, symbol processing engine etc.

In this demo example we're going to create an interface *Expression* and concrete classes implementing the *Expression*interface. A class *TerminalExpression* is defined which acts as a main interpreter of context in question. Other classes *OrExpression*, *AndExpression* are used to create combinational expressions. *InterpreterDemo*, our demo class will use *Expression* class to create rules and demonstrate parsing of expressions.

Class Diagram:rver

Type here to search 28°C ENG 18:23 03-08-2023


M2006C3MII
6:25 PM
InterpreterDemo
+getMaleExpression() : void
+getmarriedWomenexpression()
: void
uses
Expression
<<Interface>>
+interpret() : void
implements
implement
Type here to search
28°C
ENG
18:24
03-08-2023


M2006C3MII
6:25 PM
uses
Expression
<<Interface>>
+interpret() : void
implements
implement
TerminalExpression
-data : String
+TerminalExpression()
+interpret(): boolean
AndExpression
-expr1 : Expression
-expr2 : Expression
+AndExpression()
+interpret(): boolean
OrExpression
-expr1 : Expression
-expr2 : Expression
+OrExpression()
+interpret(): boolean
Type here to search
28°C
ENG
18:24
03-08-2023

M2006C3MII

6:26 PM

Steps

Use the following steps to implement the above mentioned design pattern.

Step 1

Create an expression interface.

Expression.java

```
public interface Expression {
public boolean interpret(String context);
}
```

Step 2

Create concrete classes implementing the above interface.

TerminalExpression.java

```
public class TerminalExpression implements Expression {
private String data;
public TerminalExpression(String data){
this.data = data;
}
@Override
public boolean interpret(String context) {
if(context.contains(data)){
return true;
}
return false;
```

Type here to search

28°C ENG 18:24 03-08-2023

M2006C3MII

6:26 PM

Step 2

Create concrete classes implementing the above interface.

TerminalExpression.java

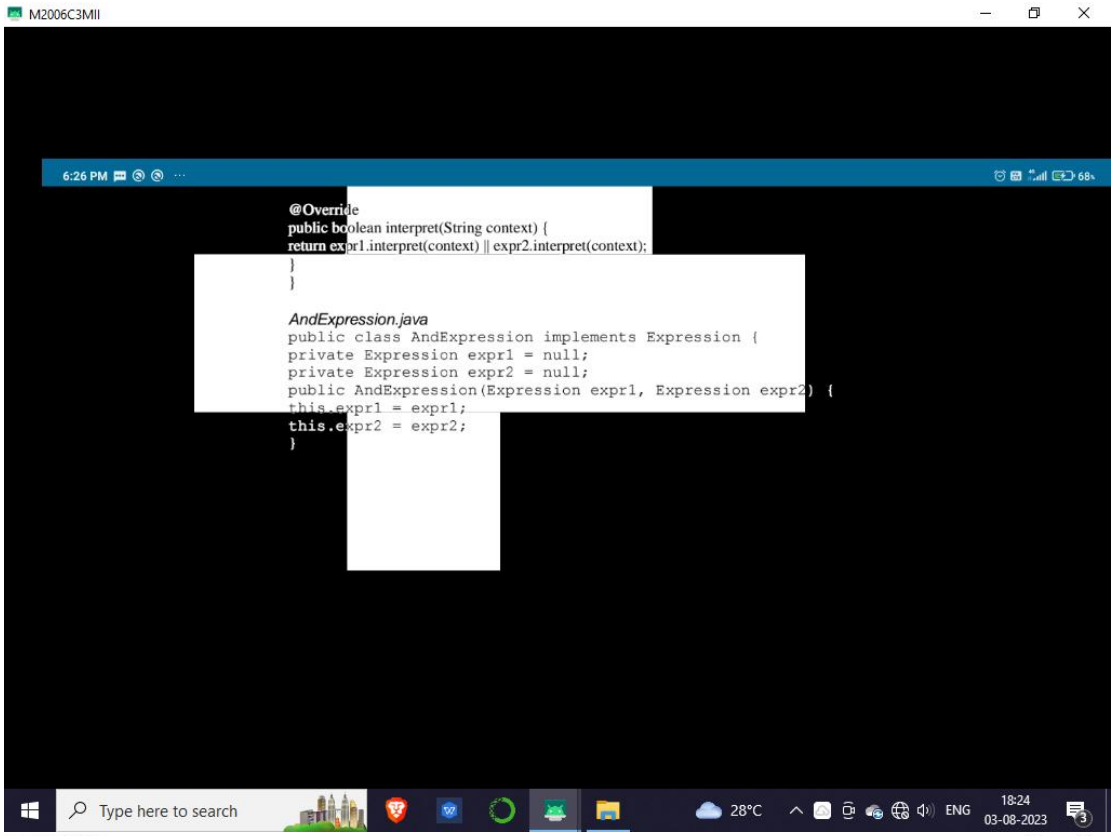
```
public class TerminalExpression implements Expression {
private String data;
public TerminalExpression(String data){
this.data = data;
}
@Override
public boolean interpret(String context) {
if(context.contains(data)){
return true;
}
return false;
}
}
```

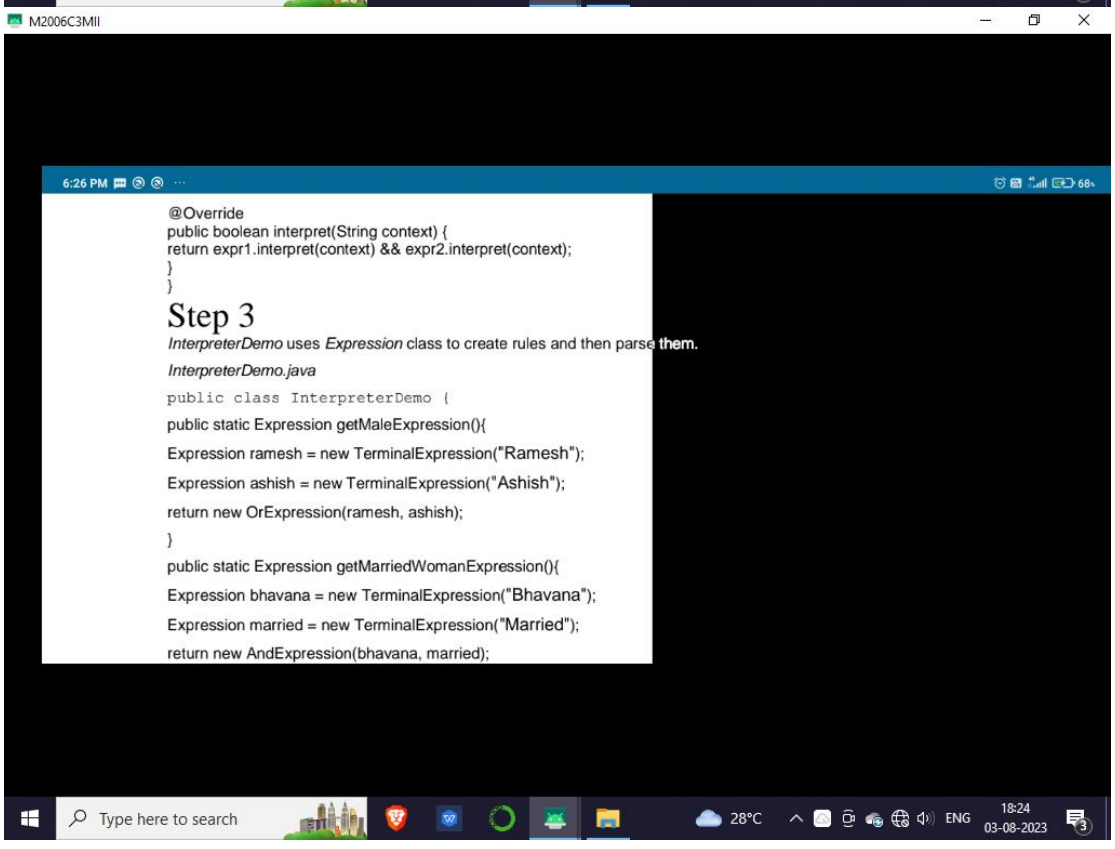
OrExpression.java

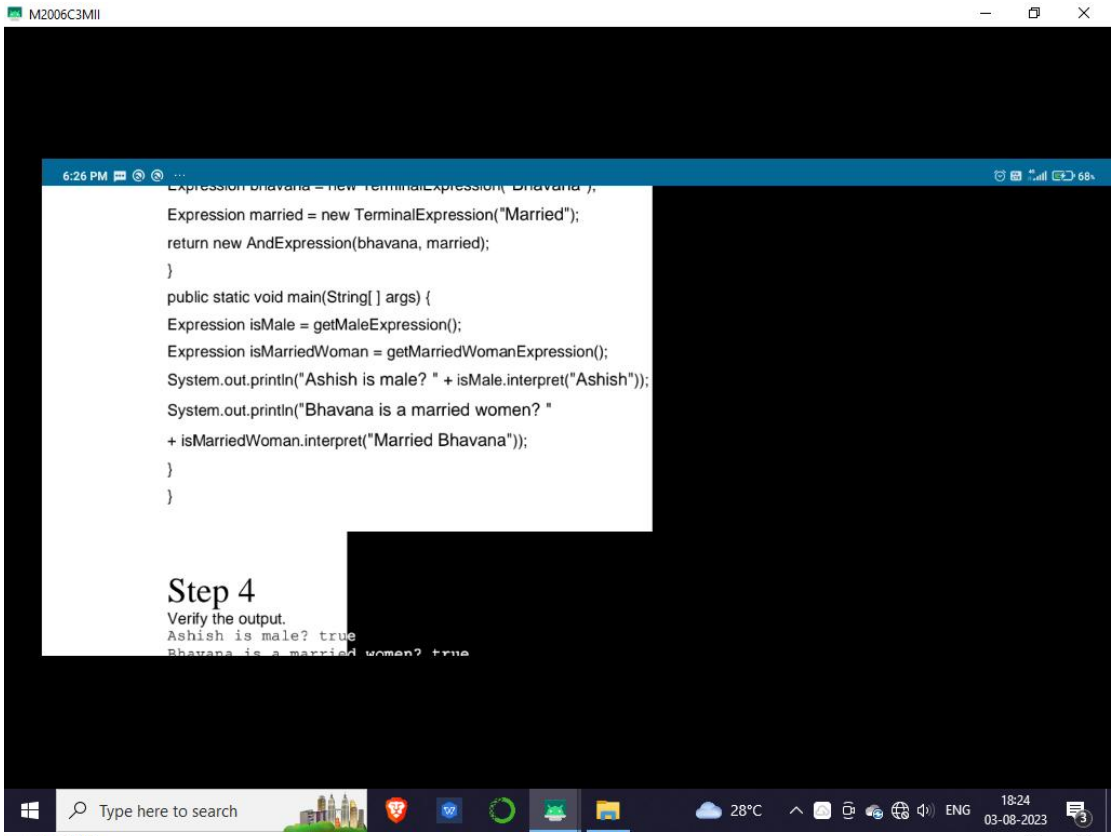
```
public class OrExpression implements
Expression {
private Expression expr1 = null;
private Expression expr2 = null;
public OrExpression(Expression expr1, Expression expr2) {
this.expr1 = expr1;
this.expr2 = expr2;
}
```

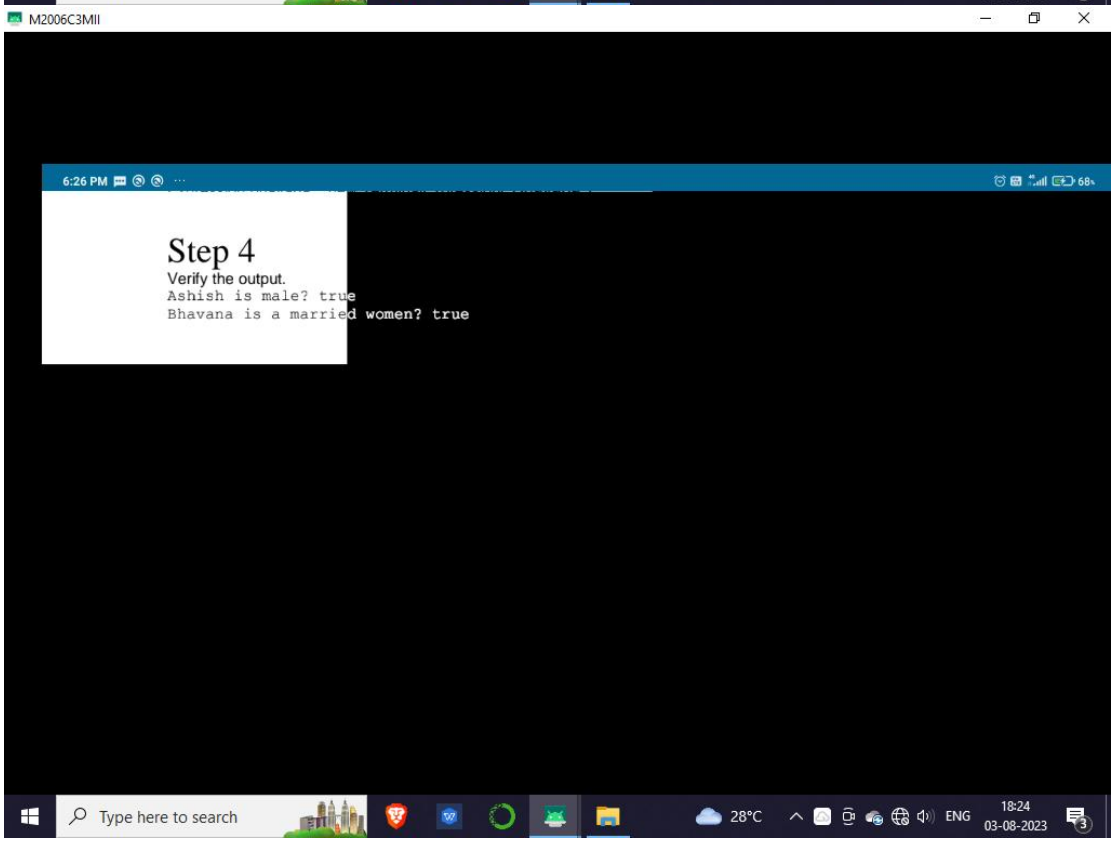
Type here to search

28°C ENG 18:24 03-08-2023


M2006C3MII
6:26 PM
@Override
public boolean interpret(String context) {
return expr1.interpret(context) || expr2.interpret(context);
}
}
AndExpression.java
public class AndExpression implements Expression {
private Expression expr1 = null;
private Expression expr2 = null;
public AndExpression(Expression expr1, Expression expr2) {
this.expr1 = expr1;
this.expr2 = expr2;
}
Type here to search
28°C
ENG
18:24
03-08-2023


M2006C3MII
6:26 PM
@Override
public boolean interpret(String context) {
return expr1.interpret(context) && expr2.interpret(context);
}
}
Step 3
InterpreterDemo uses Expression class to create rules and then parse them.
InterpreterDemo.java
public class InterpreterDemo {
public static Expression getMaleExpression(){
Expression ramesh = new TerminalExpression("Ramesh");
Expression ashish = new TerminalExpression("Ashish");
return new OrExpression(ramesh, ashish);
}
public static Expression getMarriedWomanExpression(){
Expression bhavana = new TerminalExpression("Bhavana");
Expression married = new TerminalExpression("Married");
return new AndExpression(bhavana, married);
Type here to search
28°C
ENG
18:24
03-08-2023


M2006C3MII
6:26 PM
Expression married = new TerminalExpression("Married");
return new AndExpression(bhavana, married);
}
public static void main(String[ ] args) {
Expression isMale = getMaleExpression();
Expression isMarriedWoman = getMarriedWomanExpression();
System.out.println("Ashish is male? " + isMale.interpret("Ashish"));
System.out.println("Bhavana is a married women? "
+ isMarriedWoman.interpret("Married Bhavana"));
}
}
Step 4
Verify the output.
Ashish is male? true
Type here to search
28°C
ENG
18:24
03-08-2023


M2006C3MII
6:26 PM
Step 4
Verify the output.
Ashish is male? true
Bhavana is a married women? true
Type here to search
28°C
ENG
18:24
03-08-2023

M2006C3MII

6:26 PM

Applicability & Examples

The iterator pattern allow us to:

- access contents of a collection without exposing its internal structure.
- support multiple simultaneous traversals of a collection.
- provide a uniform interface for traversing different collection.

Example 1: This exmple is using a collection of books and it uses an iterator to iterate through the collection. The main actors are:

- **IIterator** - This interface represent the AbstractIterator, defining the iterator
- **BookIterator** - This is the implementation of Iterator(implements the IIterator interface)
- **IContainer** - This is an interface defining the Agregate
- **BooksCollection** - An implementation of the collection

Here is the code for the abstractions IIterator and IContainer:

Type here to search

28°C ENG 18:24 03-08-2023

M2006C3MII

6:26 PM

Here is the code for the abstractions IIterator and IContainer:

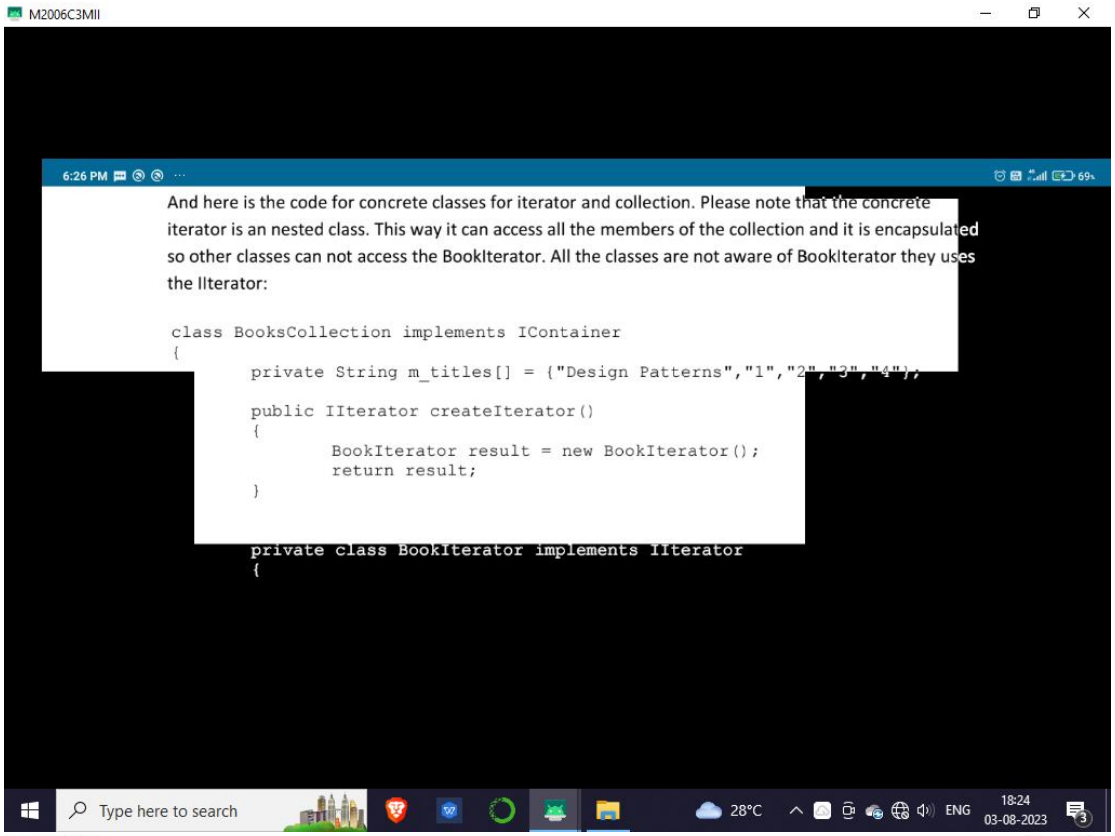
```
interface IIterator
{
        public boolean hasNext();
        public Object next();
}

interface IContainer
{
        public IIterator createIterator();
}
```

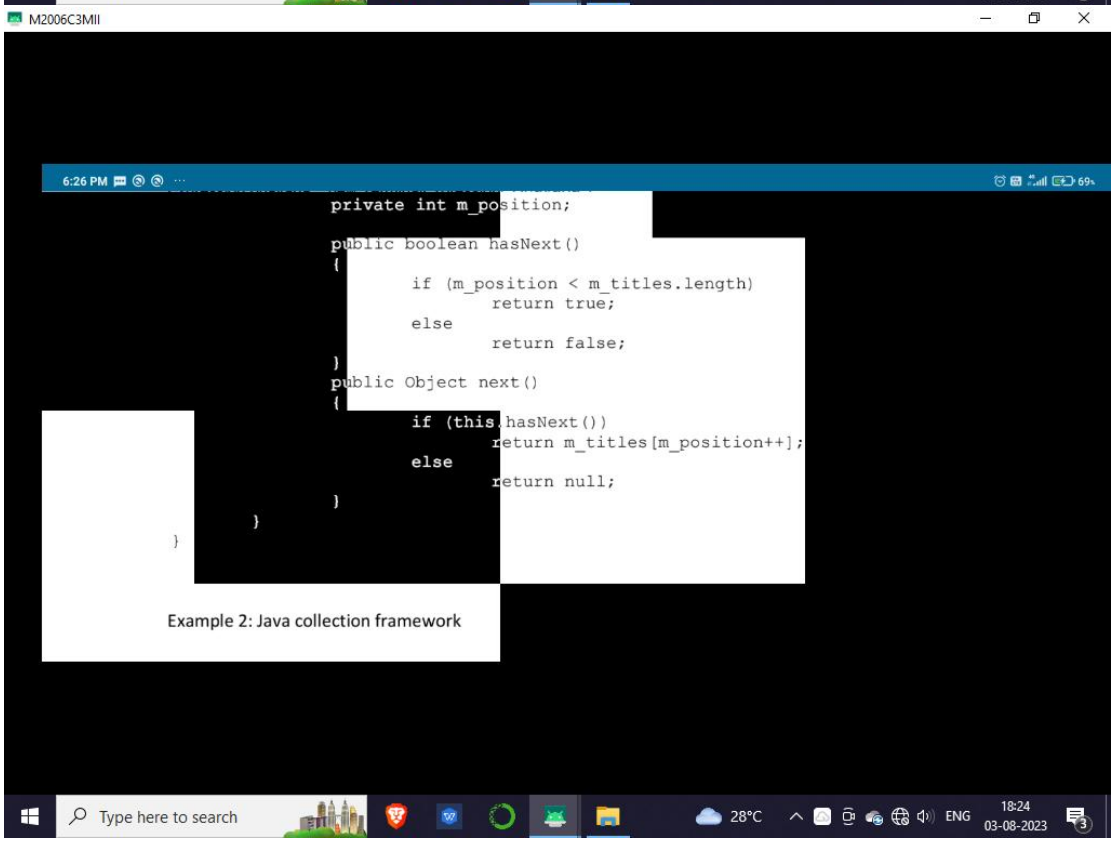
And here is the code for concrete classes for iterator and collection. Please note that the concrete iterator is an nested class. This way it can access all the members of the collection and it is encapsulated so other classes can not access the BookIterator. All the classes are not aware of BookIterator they uses the IIterator:

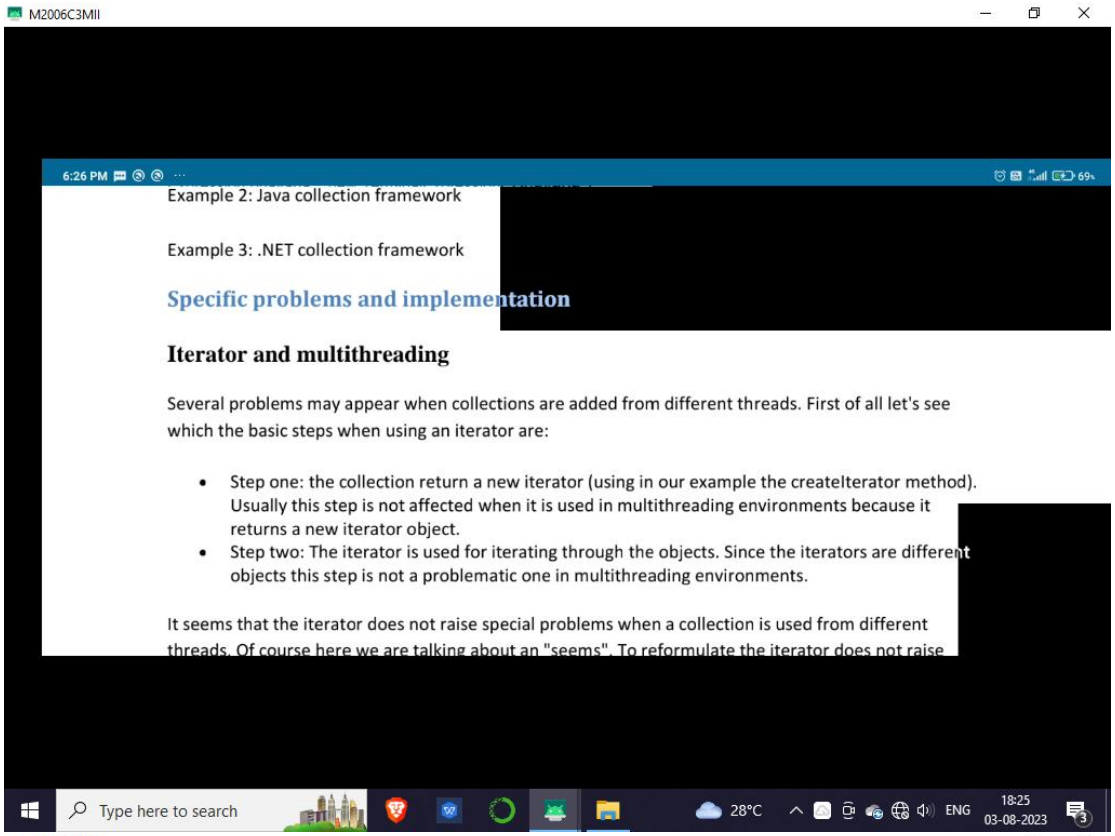
```
class BooksCollection implements IContainer
{
```

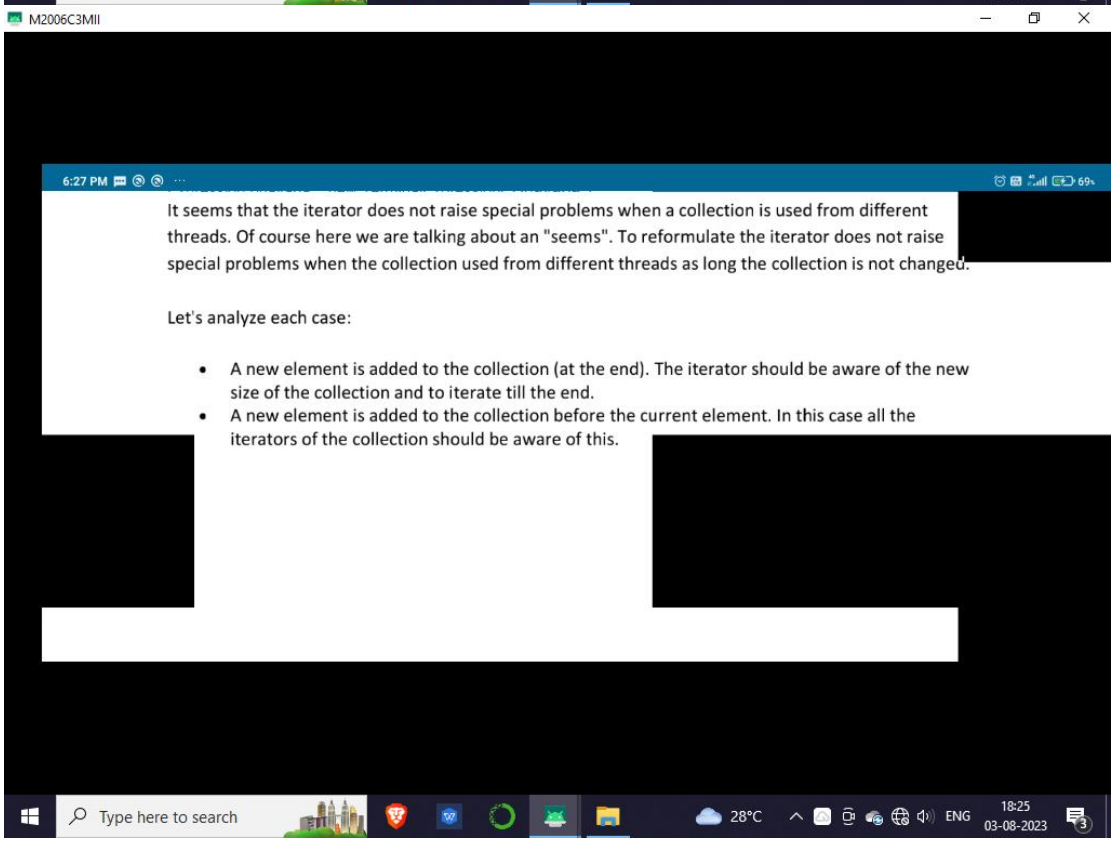
Type here to search

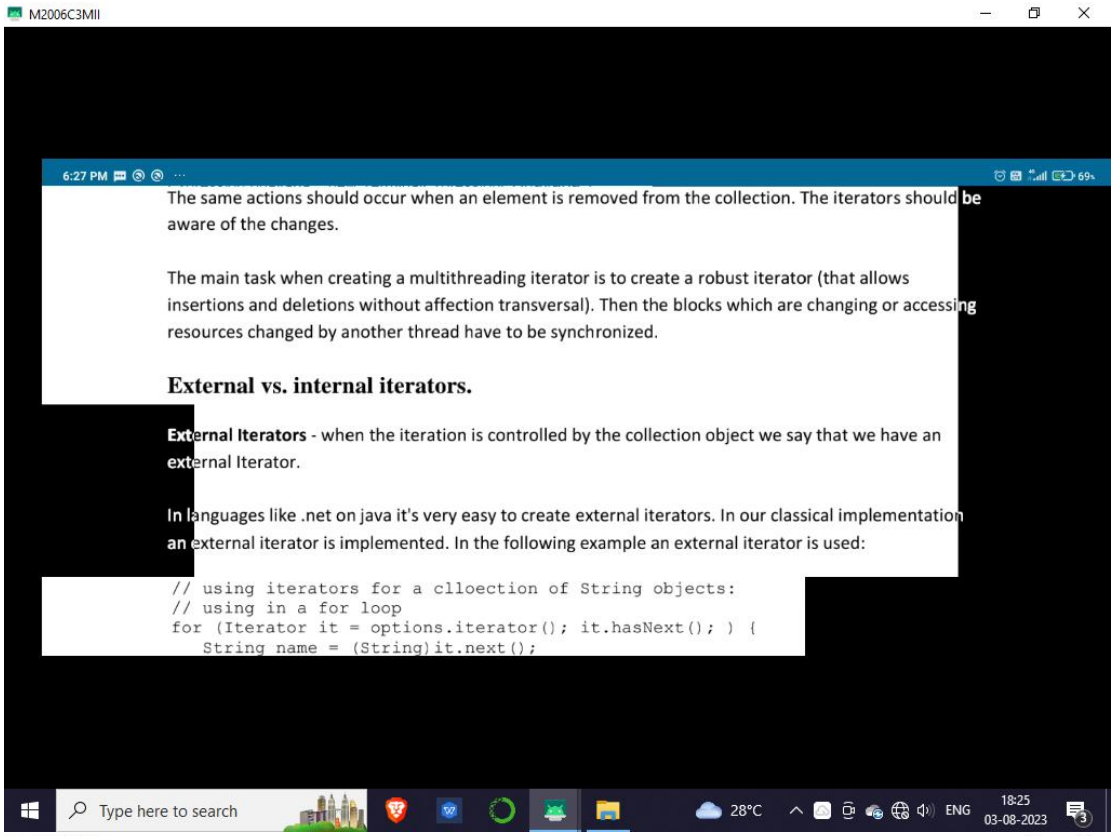
28°C ENG 18:24 03-08-2023

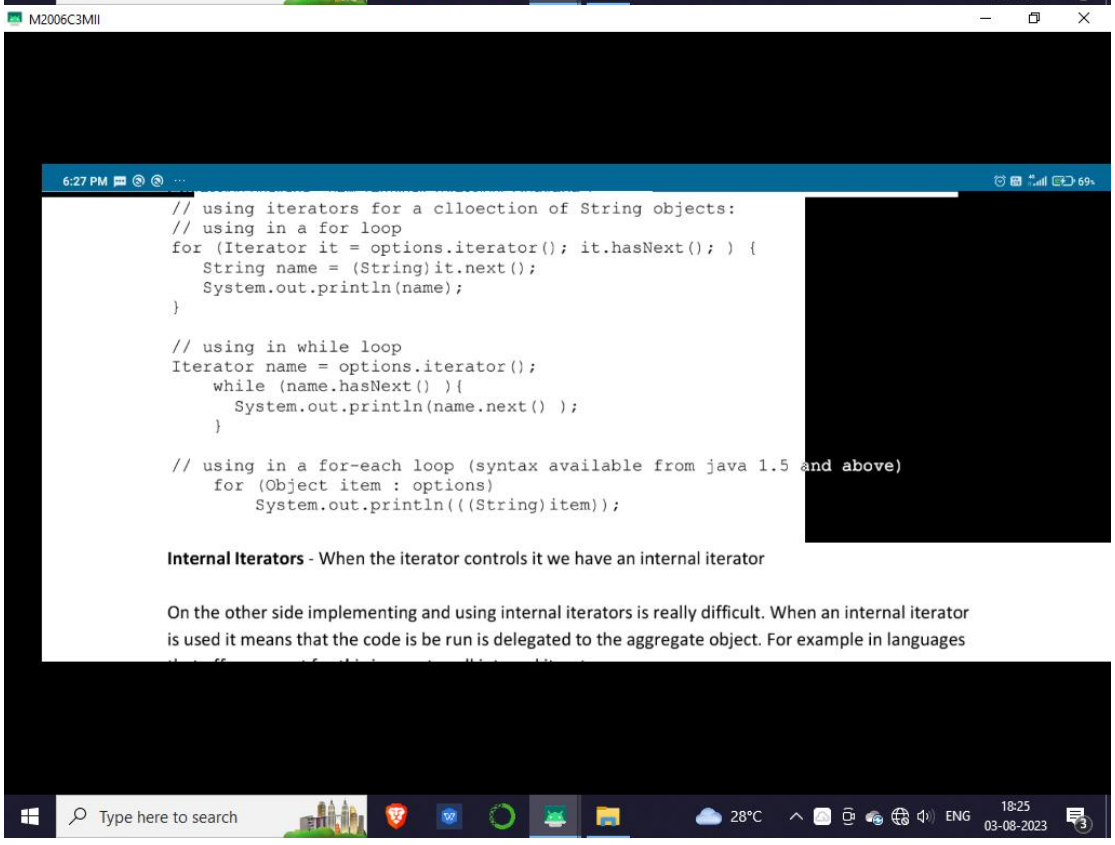

M2006C3MII
6:26 PM
And here is the code for concrete classes for iterator and collection. Please note that the concrete iterator is an nested class. This way it can access all the members of the collection and it is encapsulated so other classes can not access the BookIterator. All the classes are not aware of BookIterator they uses the IIterator:
class BooksCollection implements IContainer
{
private String m_titles[] = {"Design Patterns","1","2","3","4"};
public IIterator createIterator()
{
BookIterator result = new BookIterator();
return result;
}
private class BookIterator implements IIterator
{
Type here to search
28°C
ENG
18:24
03-08-2023

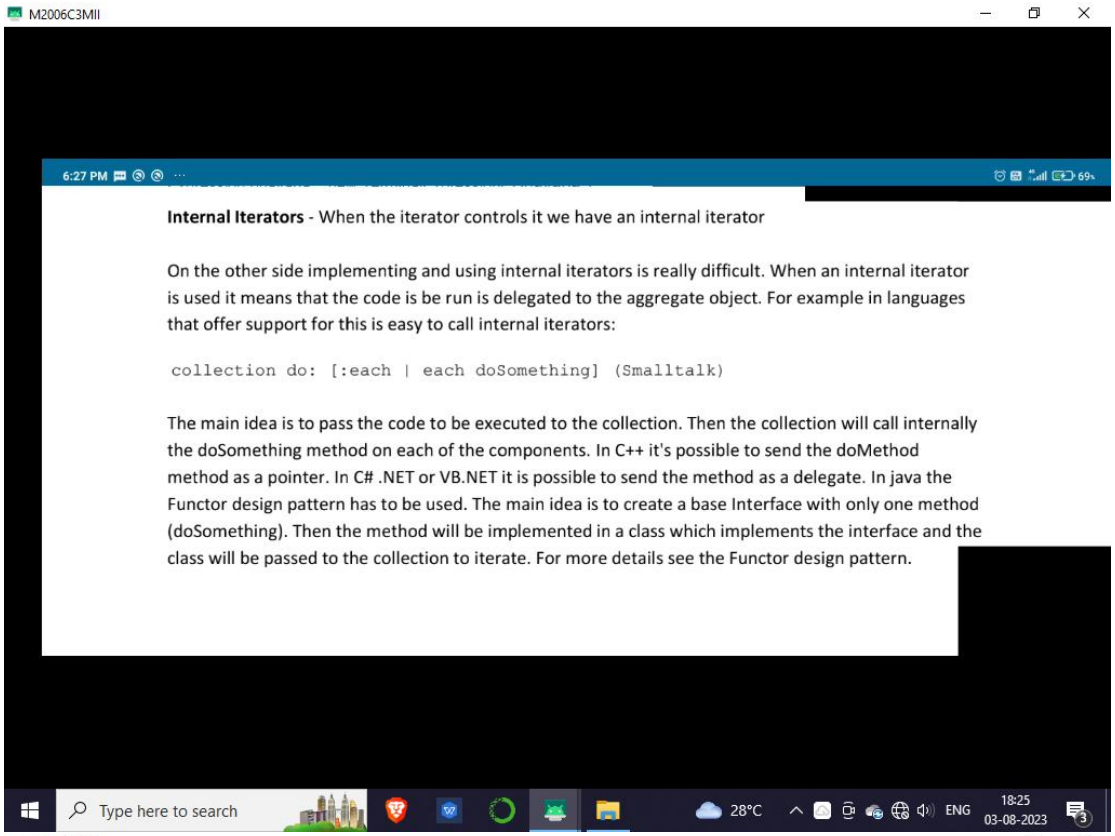

M2006C3MII
6:26 PM
private int m_position;
public boolean hasNext()
{
if (m_position < m_titles.length)
return true;
else
return false;
}
public Object next()
{
if (this.hasNext())
return m_titles[m_position++];
else
return null;
}
}
Example 2: Java collection framework
Type here to search
28°C
ENG
18:24
03-08-2023

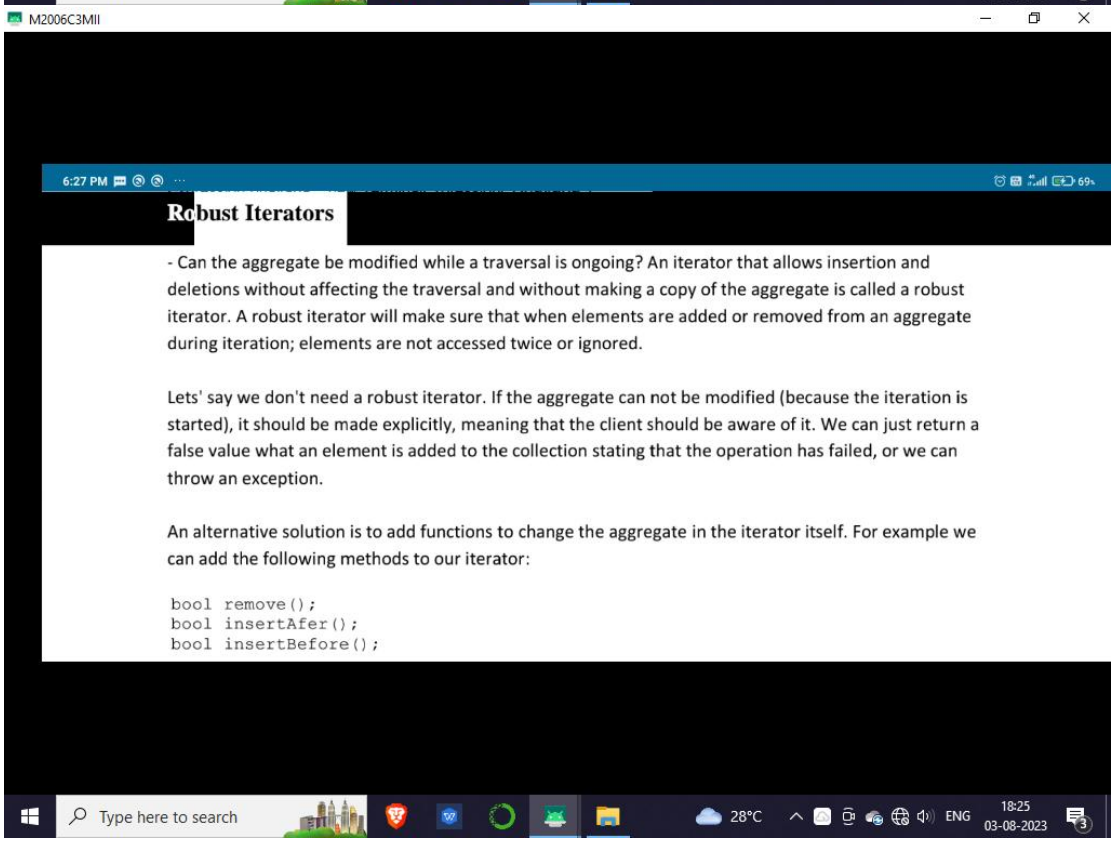

M2006C3MII
6:26 PM
Example 2: Java collection framework
Example 3: .NET collection framework
Specific problems and implementation
Iterator and multithreading
Several problems may appear when collections are added from different threads. First of all let's see which the basic steps when using an iterator are:
Step one: the collection return a new iterator (using in our example the createIterator method). Usually this step is not affected when it is used in multithreading environments because it returns a new iterator object.
Step two: The iterator is used for iterating through the objects. Since the iterators are different objects this step is not a problematic one in multithreading environments.
It seems that the iterator does not raise special problems when a collection is used from different threads. Of course here we are talking about an "seems". To reformulate the iterator does not raise
Type here to search
28°C
ENG
18:25
03-08-2023

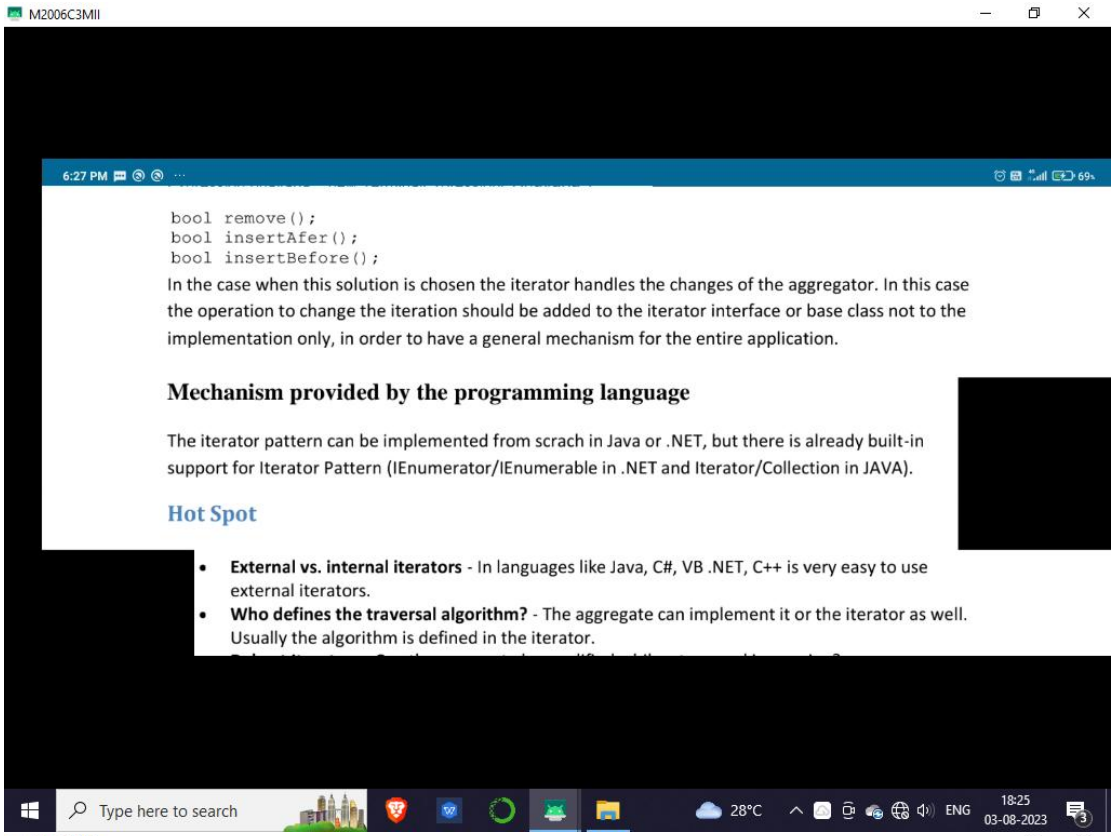

M2006C3MII
6:27 PM
It seems that the iterator does not raise special problems when a collection is used from different threads. Of course here we are talking about an "seems". To reformulate the iterator does not raise special problems when the collection used from different threads as long the collection is not changed.
Let's analyze each case:
A new element is added to the collection (at the end). The iterator should be aware of the new size of the collection and to iterate till the end.
A new element is added to the collection before the current element. In this case all the iterators of the collection should be aware of this.
Type here to search
28°C
ENG
18:25
03-08-2023

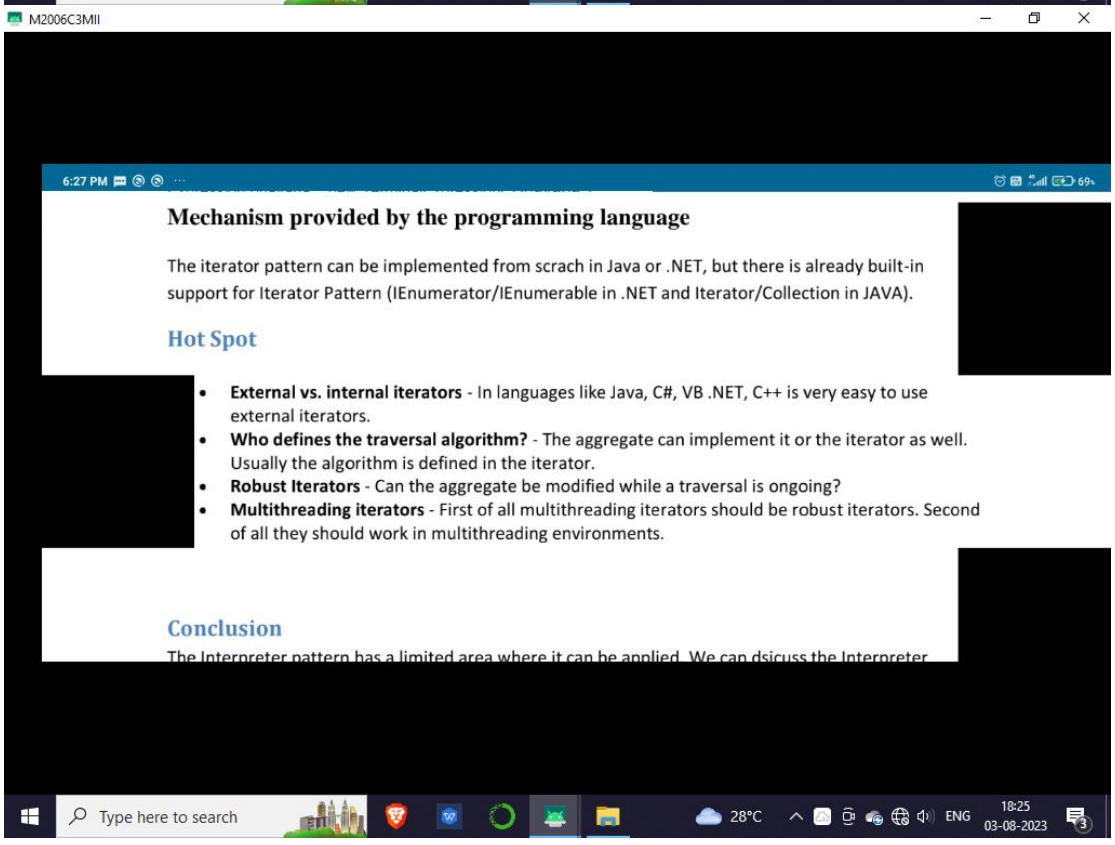

M2006C3MII
6:27 PM
The same actions should occur when an element is removed from the collection. The iterators should be aware of the changes.
The main task when creating a multithreading iterator is to create a robust iterator (that allows insertions and deletions without affection transversal). Then the blocks which are changing or accessing resources changed by another thread have to be synchronized.
External vs. internal iterators.
External Iterators - when the iteration is controlled by the collection object we say that we have an external Iterator.
In languages like .net on java it's very easy to create external iterators. In our classical implementation an external iterator is implemented. In the following example an external iterator is used:
// using iterators for a clloection of String objects:
// using in a for loop
for (Iterator it = options.iterator(); it.hasNext();) {
String name = (String)it.next();
Type here to search
28°C
ENG
18:25
03-08-2023


M2006C3MII
6:27 PM
// using iterators for a clloection of String objects:
// using in a for loop
for (Iterator it = options.iterator(); it.hasNext();) {
String name = (String)it.next();
System.out.println(name);
}
// using in while loop
Iterator name = options.iterator();
while (name.hasNext()){
System.out.println(name.next());
}
// using in a for-each loop (syntax available from java 1.5 and above)
for (Object item : options)
System.out.println(((String)item));
Internal Iterators - When the iterator controls it we have an internal iterator
On the other side implementing and using internal iterators is really difficult. When an internal iterator is used it means that the code is be run is delegated to the aggregate object. For example in languages
Type here to search
28°C
ENG
18:25
03-08-2023


M2006C3MII
6:27 PM
Internal Iterators - When the iterator controls it we have an internal iterator
On the other side implementing and using internal iterators is really difficult. When an internal iterator is used it means that the code is be run is delegated to the aggregate object. For example in languages that offer support for this is easy to call internal iterators:
collection do: [:each | each doSomething] (Smalltalk)
The main idea is to pass the code to be executed to the collection. Then the collection will call internally the doSomething method on each of the components. In C++ it's possible to send the doMethod method as a pointer. In C# .NET or VB.NET it is possible to send the method as a delegate. In java the Functor design pattern has to be used. The main idea is to create a base Interface with only one method (doSomething). Then the method will be implemented in a class which implements the interface and the class will be passed to the collection to iterate. For more details see the Functor design pattern.
Type here to search
28°C
ENG
18:25
03-08-2023


M2006C3MII
6:27 PM
Robust Iterators
- Can the aggregate be modified while a traversal is ongoing? An iterator that allows insertion and deletions without affecting the traversal and without making a copy of the aggregate is called a robust iterator. A robust iterator will make sure that when elements are added or removed from an aggregate during iteration; elements are not accessed twice or ignored.
Lets' say we don't need a robust iterator. If the aggregate can not be modified (because the iteration is started), it should be made explicitly, meaning that the client should be aware of it. We can just return a false value what an element is added to the collection stating that the operation has failed, or we can throw an exception.
An alternative solution is to add functions to change the aggregate in the iterator itself. For example we can add the following methods to our iterator:
bool remove();
bool insertAfer();
bool insertBefore();
Type here to search
28°C
ENG
18:25
03-08-2023


M2006C3MII
6:27 PM
bool remove();
bool insertAfer();
bool insertBefore();
In the case when this solution is chosen the iterator handles the changes of the aggregator. In this case the operation to change the iteration should be added to the iterator interface or base class not to the implementation only, in order to have a general mechanism for the entire application.
Mechanism provided by the programming language
The iterator pattern can be implemented from scrach in Java or .NET, but there is already built-in support for Iterator Pattern (IEnumerator/IEnumerable in .NET and Iterator/Collection in JAVA).
Hot Spot
External vs. internal iterators - In languages like Java, C#, VB .NET, C++ is very easy to use external iterators.
Who defines the traversal algorithm? - The aggregate can implement it or the iterator as well. Usually the algorithm is defined in the iterator.
Type here to search
28°C
ENG
18:25
03-08-2023


M2006C3MII
6:27 PM
Mechanism provided by the programming language
The iterator pattern can be implemented from scrach in Java or .NET, but there is already built-in support for Iterator Pattern (IEnumerator/IEnumerable in .NET and Iterator/Collection in JAVA).
Hot Spot
External vs. internal iterators - In languages like Java, C#, VB .NET, C++ is very easy to use external iterators.
Who defines the traversal algorithm? - The aggregate can implement it or the iterator as well. Usually the algorithm is defined in the iterator.
Robust Iterators - Can the aggregate be modified while a traversal is ongoing?
Multithreading iterators - First of all multithreading iterators should be robust iterators. Second of all they should work in multithreading environments.
Conclusion
The Interpreter pattern has a limited area where it can be applied. We can dsicuss the Interpreter
Type here to search
28°C
ENG
18:25
03-08-2023

M2006C3MII

6:27 PM

Conclusion

The Interpreter pattern has a limited area where it can be applied. We can dsicuss the Interpreter pattern only in terms of formal grammars but in this area there are better solutions and this is the reason why this pattern is not so frequenlty used. This pattern can be applied for parssing light expressions defined in simple grammars and sometimes in simple rule engines.

Type here to search 28°C ENG 18:25 03-08-2023

M2006C3MII

6:27 PM

Mediator Pattern

Motivation

In order to have a good object oriented design we have to create lots of classes interacting one with each other. If certain principles are not applied the final framework will end in a total mess where each object relies on many other objects in order to run. In order to avoid tight coupled frameworks, we need a mechanism to facilitate the interaction between objects in a manner in that objects are not aware of the existence of other objects.

Let's take the example of a screen. When we create it we add all sort of controls to the screen. This control need to interact with all the other control. For example when a button is pressed it must know if the data is valid in other controls. As you have seen if you created different applications using forms you don't have to modify each control class each time you add a new control to the form. All the operations between controls are managed by the form class itself. This class is called mediator.

Intent

Type here to search 28°C ENG 18:25 03-08-2023

M2006C3MII

6:27 PM

Intent

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

Implementation

Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintainability of the code by loose coupling. Mediator pattern falls under behavioral pattern category.

Implementation

In this example We're elaborating mediator pattern by example of a Chat Room where multiple users can send message to Chat Room and it is the responsibility of Chat Room to show the messages to all users. We've created two classes *ChatRoom* and *User*. *User* objects will use *ChatRoom* method to share their messages. *MediatorPatternDemo*, our demo class will use *User* objects to show communication between them

Type here to search 28°C ENG 18:25 03-08-2023

M2006C3MII

6:27 PM

Implementation

Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintainability of the code by loose coupling. Mediator pattern falls under behavioral pattern category.

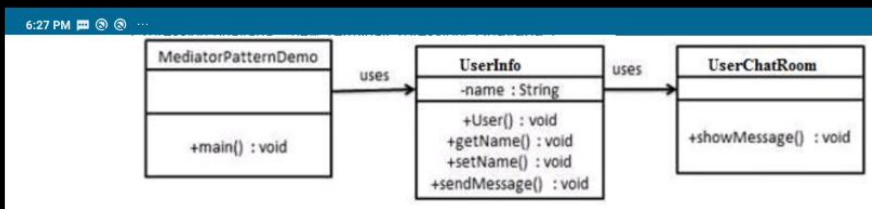
Implementation

In this example We're elaborating mediator pattern by example of a Chat Room where multiple users can send message to Chat Room and it is the responsibility of Chat Room to show the messages to all users. We've created two classes *ChatRoom* and *User*. *User* objects will use *ChatRoom* method to share their messages. *MediatorPatternDemo*, our demo class will use *User* objects to show communication between them

Class Diagram:

Type here to search 28°C ENG 18:25 03-08-2023

M2006C3MII



Steps

Use the following steps to implement the above mentioned design pattern.

Step 1

Create mediator class.

ChatRoom.java

```
import java.util.Date;
public class ChatRoom {
public static void showMessage(UserInfo user, String message){
System.out.println(new Date().toString()
+ " [" + user.getName() +"] : " + message);
}
}
```

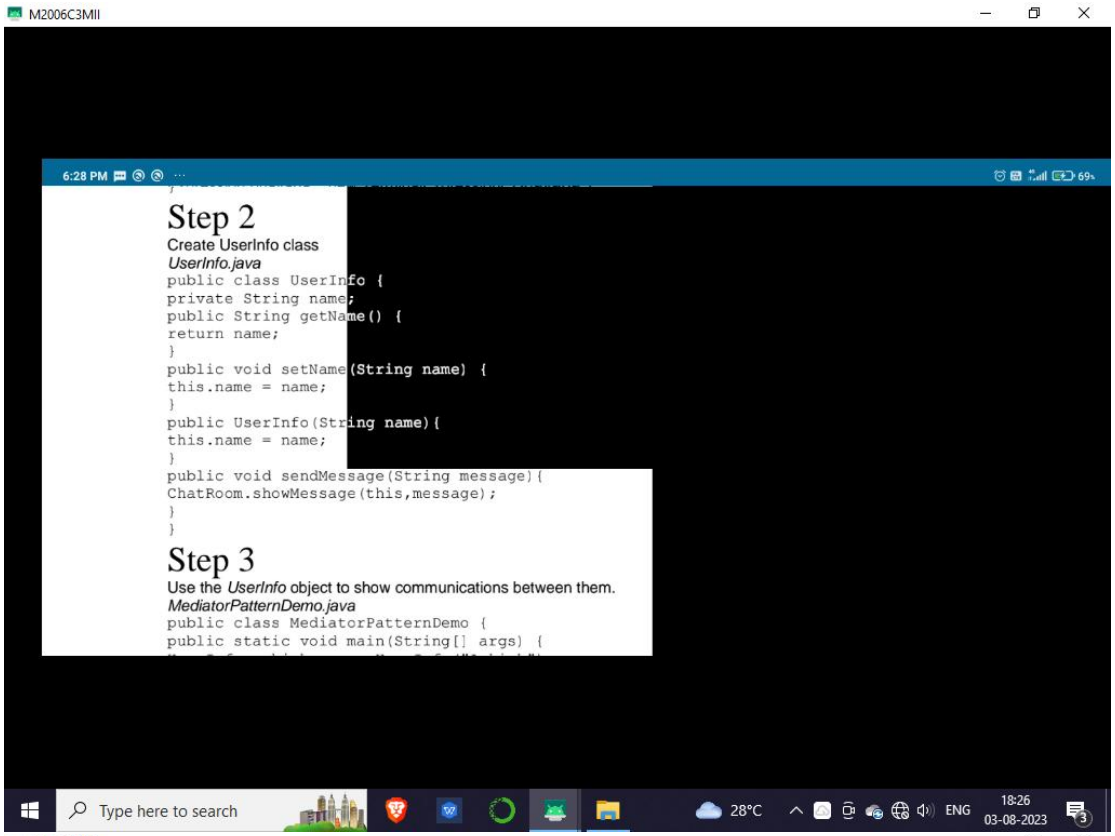
M2006C3MII

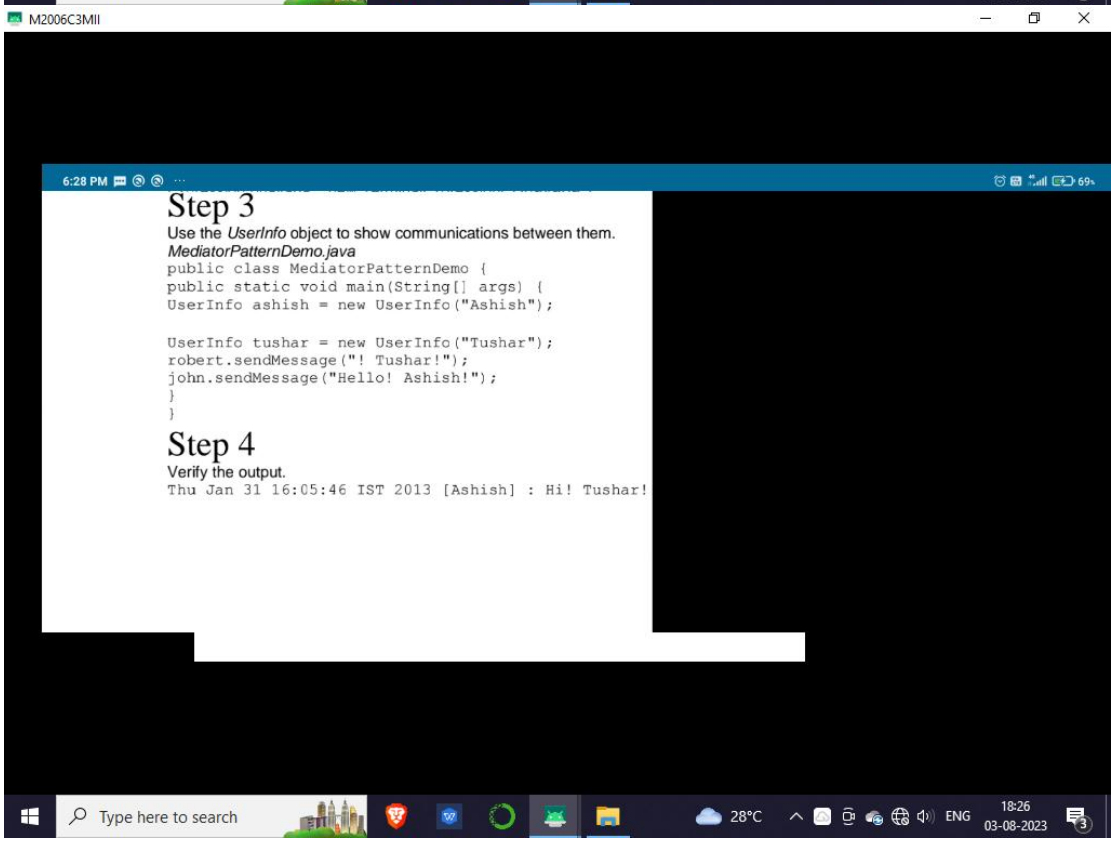
Step 2

Create UserInfo class

UserInfo.java

```
public class UserInfo {
private String name;
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
```


M2006C3MII
6:28 PM
Step 2
Create UserInfo class
UserInfo.java
public class UserInfo {
private String name;
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public UserInfo(String name){
this.name = name;
}
public void sendMessage(String message){
ChatRoom.showMessage(this,message);
}
}
Step 3
Use the UserInfo object to show communications between them.
MediatorPatternDemo.java
public class MediatorPatternDemo {
public static void main(String[] args) {
Type here to search
28°C
ENG
18:26
03-08-2023


M2006C3MII
6:28 PM
Step 3
Use the UserInfo object to show communications between them.
MediatorPatternDemo.java
public class MediatorPatternDemo {
public static void main(String[] args) {
UserInfo ashish = new UserInfo("Ashish");
UserInfo tushar = new UserInfo("Tushar");
robert.sendMessage("! Tushar!");
john.sendMessage("Hello! Ashish!");
}
}
Step 4
Verify the output.
Thu Jan 31 16:05:46 IST 2013 [Ashish] : Hi! Tushar!
Type here to search
28°C
ENG
18:26
03-08-2023

M2006C3MII

6:28 PM

Thu Jan 31 16:05:46 IST 2013 [Tushar] : Hello! Ashish!

Applicability

According to Gamma et al, the Mediator pattern should be used when:

- a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- reusing an object is difficult because it refers to and communicates with many other objects.
- a behavior that's distributed between several classes should be customizable without a lot of subclassing.

Examples:

Example 1 - GUI Libraries

The mediator example is one pattern that is already used in many applications. One of the

Type here to search 28°C ENG 18:26 03-08-2023

M2006C3MII

6:28 PM

Example 1 - GUI Libraries

The mediator example is one pattern that is already used in many applications. One of the examples is represented by the Dialog classes in GUI applications frameworks. A Dialog window is a collection of graphic and non-graphic controls. The Dialog class provides the mechanism to facilitate the interaction between controls. For example, when a new value is selected from a ComboBox object a Label has to display a new value. Both the ComboBox and the Label are not aware of each other structure and all the interaction is managed by the Dialog object. Each control is not aware of the existence of other controls.

Specific problems and implementation

Abstract Mediators

There is no need to create an Abstract Mediator class or an interface as long as the colleagues are going to use only one mediator. The definition of an abstract Mediator is required only if the

Type here to search 28°C ENG 18:26 03-08-2023

M2006C3MII

6:28 PM

Abstract Mediators

There is no need to create an Abstract Mediator class or an interface as long as the colleagues are going to use only one mediator. The definition of an abstract Mediator is required only if the colleagues needs to work with different mediators.

Communication between mediators and colleagues

There are different ways to realize the communication between the colleagues and its mediator:

- One of the most used methods is to use the Observer pattern. The mediator can be also an observer and the Colleagues can be implement an observable object. Each time an change is made in the state of the observable object, the observer(mediator) gets notified and it notify all other colleagues objects.
- Alternative methods can be used to send messages to the mediator. For example a simple delegation can be used and specialised methods can be exposed by the mediator

Type here to search 28°C ENG 18:26 03-08-2023

M2006C3MII

6:28 PM

- In more complex implementations asynchronous messages can be added to to a message queue, from where they can be picked up by the mediator object

Complexity of Mediator object

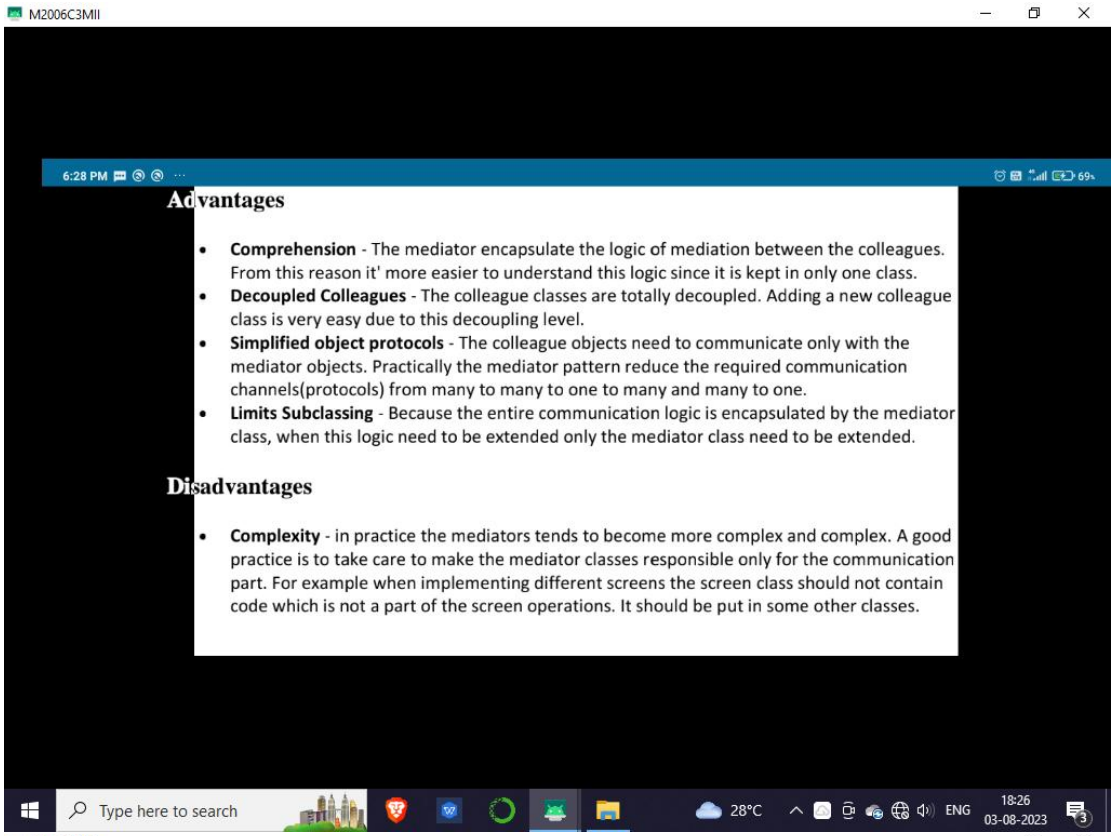
The mediator object handles all the interaction between the participants objects. One potential problem is complexity of the mediator when the number of participants is a high and the different participant classes is high. If you created custom dialogs for GUI applications you remember that after some time the dialogs classes become really complex because they had to manage a lot of operations.

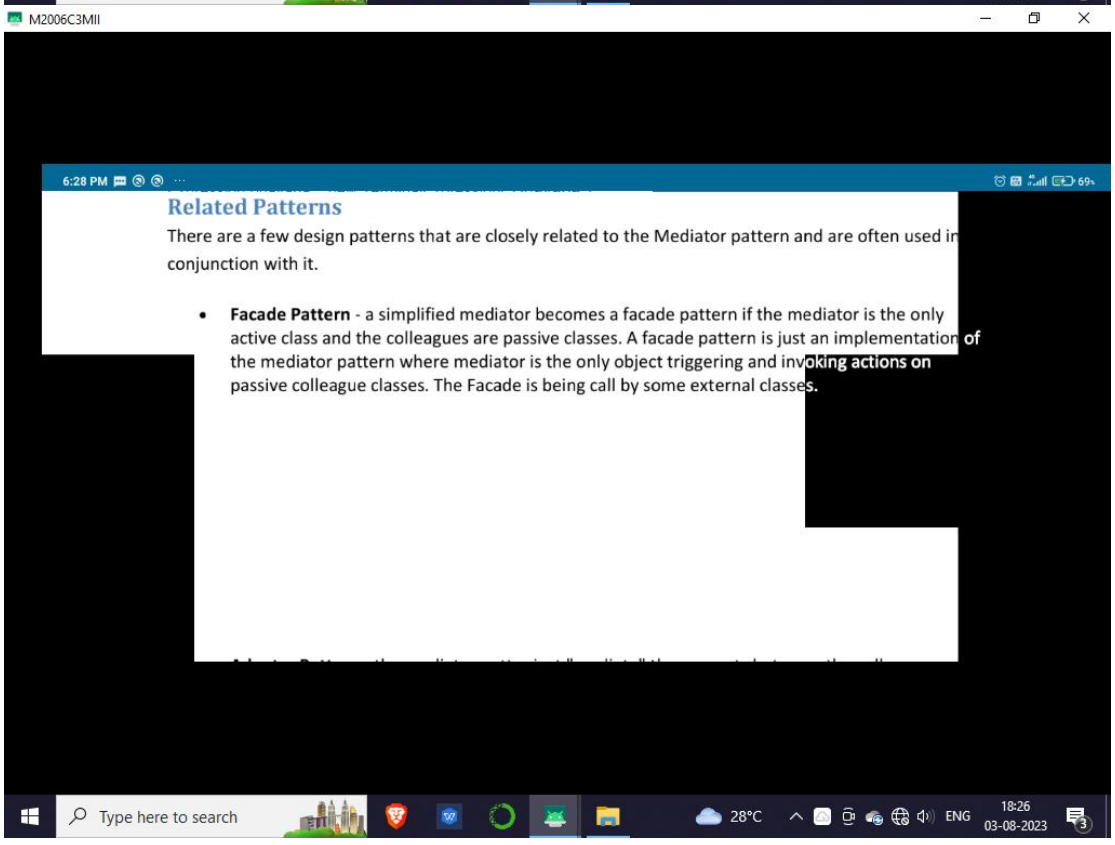
Consequences

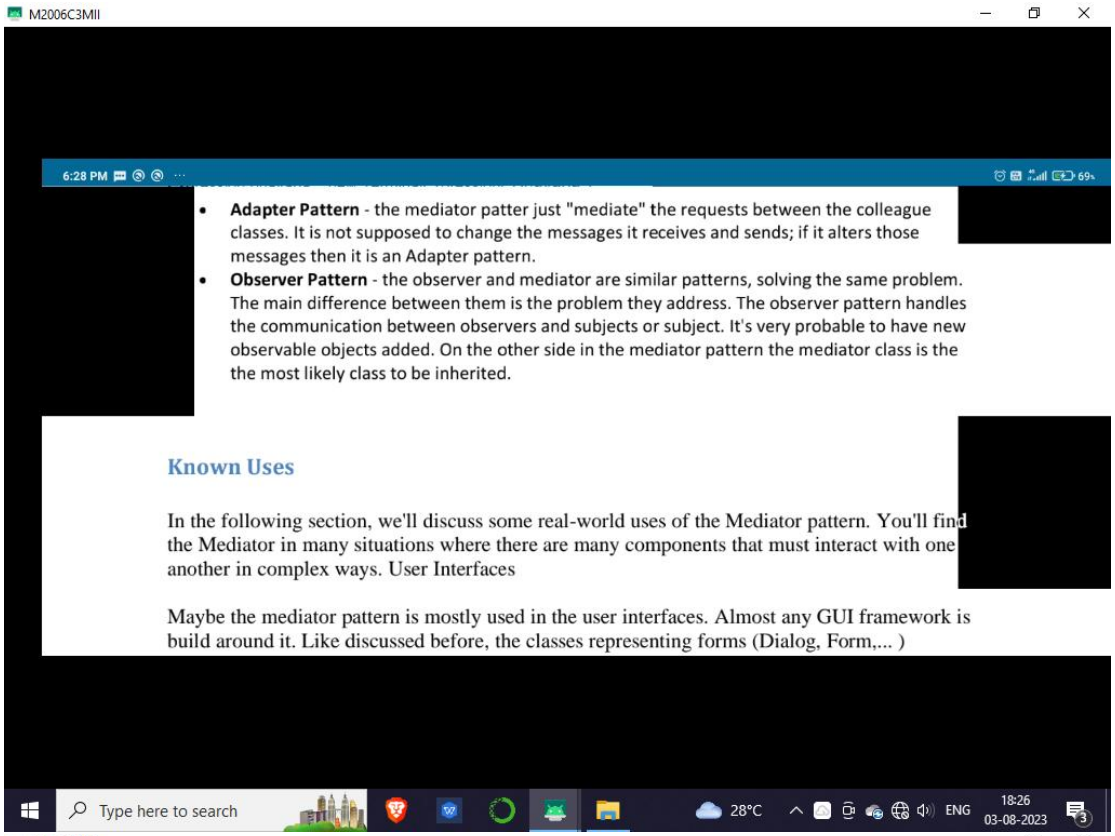
As with most design patterns, there are both advantages and disadvantages of using the Mediator Patern. The following section will briefly outline a few of these issues.

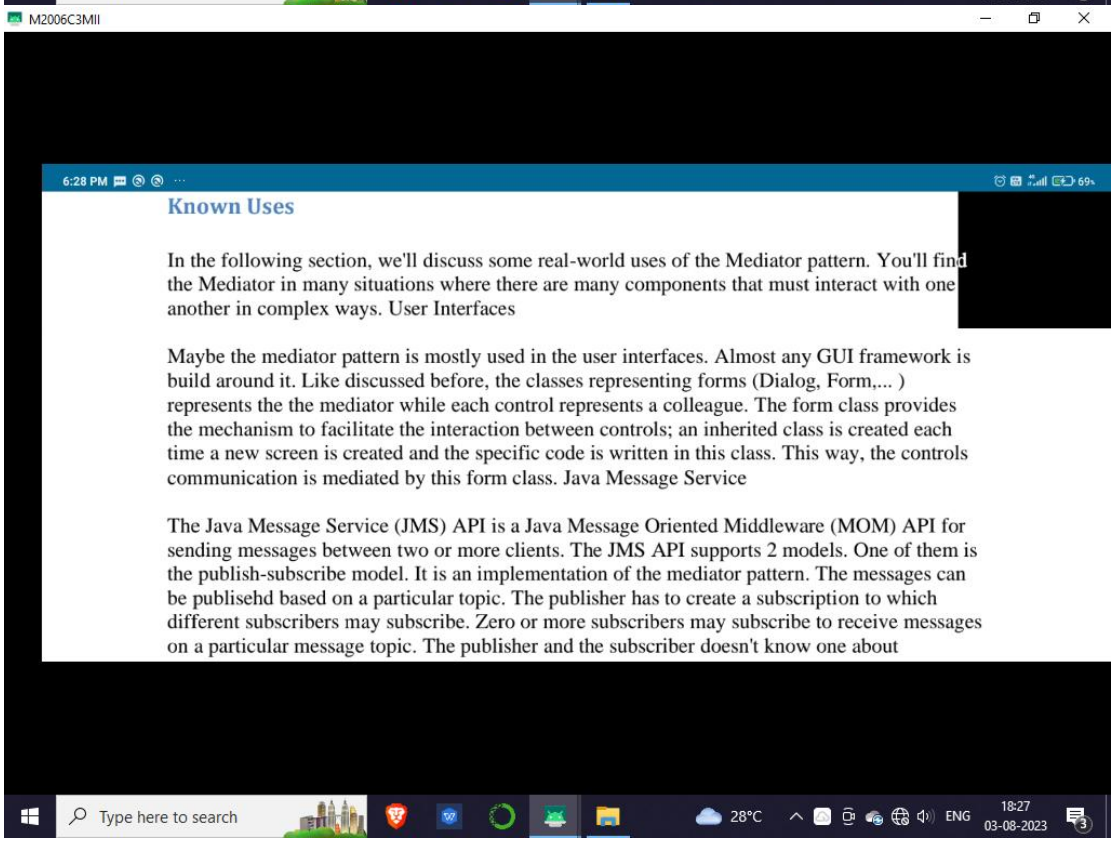
Advantages

Type here to search 28°C ENG 18:26 03-08-2023


M2006C3MII
6:28 PM
Advantages
Comprehension - The mediator encapsulate the logic of mediation between the colleagues. From this reason it' more easier to understand this logic since it is kept in only one class.
Decoupled Colleagues - The colleague classes are totally decoupled. Adding a new colleague class is very easy due to this decoupling level.
Simplified object protocols - The colleague objects need to communicate only with the mediator objects. Practically the mediator pattern reduce the required communication channels(protocols) from many to many to one to many and many to one.
Limits Subclassing - Because the entire communication logic is encapsulated by the mediator class, when this logic need to be extended only the mediator class need to be extended.
Disadvantages
Complexity - in practice the mediators tends to become more complex and complex. A good practice is to take care to make the mediator classes responsible only for the communication part. For example when implementing different screens the screen class should not contain code which is not a part of the screen operations. It should be put in some other classes.
Type here to search
28°C
ENG
18:26
03-08-2023


M2006C3MII
6:28 PM
Related Patterns
There are a few design patterns that are closely related to the Mediator pattern and are often used in conjunction with it.
Facade Pattern - a simplified mediator becomes a facade pattern if the mediator is the only active class and the colleagues are passive classes. A facade pattern is just an implementation of the mediator pattern where mediator is the only object triggering and invoking actions on passive colleague classes. The Facade is being call by some external classes.
Type here to search
28°C
ENG
18:26
03-08-2023


M2006C3MII
6:28 PM
Adapter Pattern - the mediator patter just "mediate" the requests between the colleague classes. It is not supposed to change the messages it receives and sends; if it alters those messages then it is an Adapter pattern.
Observer Pattern - the observer and mediator are similar patterns, solving the same problem. The main difference between them is the problem they address. The observer pattern handles the communication between observers and subjects or subject. It's very probable to have new observable objects added. On the other side in the mediator pattern the mediator class is the the most likely class to be inherited.
Known Uses
In the following section, we'll discuss some real-world uses of the Mediator pattern. You'll find the Mediator in many situations where there are many components that must interact with one another in complex ways. User Interfaces
Maybe the mediator pattern is mostly used in the user interfaces. Almost any GUI framework is build around it. Like discussed before, the classes representing forms (Dialog, Form,...)
Type here to search
28°C
ENG
18:26
03-08-2023


M2006C3MII
6:28 PM
Known Uses
In the following section, we'll discuss some real-world uses of the Mediator pattern. You'll find the Mediator in many situations where there are many components that must interact with one another in complex ways. User Interfaces
Maybe the mediator pattern is mostly used in the user interfaces. Almost any GUI framework is build around it. Like discussed before, the classes representing forms (Dialog, Form,...) represents the the mediator while each control represents a colleague. The form class provides the mechanism to facilitate the interaction between controls; an inherited class is created each time a new screen is created and the specific code is written in this class. This way, the controls communication is mediated by this form class. Java Message Service
The Java Message Service (JMS) API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. The JMS API supports 2 models. One of them is the publish-subscribe model. It is an implementation of the mediator pattern. The messages can be publisehd based on a particular topic. The publisher has to create a subscription to which different subscribers may subscribe. Zero or more subscribers may subscribe to receive messages on a particular message topic. The publisher and the subscriber doesn't know one about
Type here to search
28°C
ENG
18:27
03-08-2023

M2006C3MII

6:29 PM

The Java Message Service (JMS) API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. The JMS API supports 2 models. One of them is the publish-subscribe model. It is an implementation of the mediator pattern. The messages can be publisehd based on a particular topic. The publisher has to create a subscription to which different subscribers may subscribe. Zero or more subscribers may subscribe to receive messages on a particular message topic. The publisher and the subscriber doesn't know one about eachother, the subscriber can be even inactive. In this case the subscriber receives the messages when it will become active.

Conclusion

The mediator pattern is used to takes the role of a hub or router and facilitates the communication between many classes. A similarity can be made with the database systems. The mediator transform a hard to implement relation of many to many, where each calls has to communicate with each other class, in 2 relations, easy to implement, of many to one and one to many, where the communication is handled by the mediator class.

Type here to search 28°C ENG 18:27 03-08-2023

M2006C3MII

6:29 PM

Conclusion

The mediator pattern is used to takes the role of a hub or router and facilitates the communication between many classes. A similarity can be made with the database systems. The mediator transform a hard to implement relation of many to many, where each calls has to communicate with each other class, in 2 relations, easy to implement, of many to one and one to many, where the communication is handled by the mediator class.

Type here to search 28°C ENG 18:27 03-08-2023

M2006C3MII

6:29 PM

Memento Pattern

Motivation

It is sometimes necessary to capture the internal state of an object at some point and have the ability to restore the object to that state later in time. Such a case is useful in case of error or failure. Consider the case of a calculator object with an undo operation such a calculator could simply maintain a list of all previous operation that it has performed and thus would be able to restore a previous calculation it has performed. This would cause the calculator object to become larger, more complex, and heavyweight, as the calculator object would have to provide additional undo functionality and should maintain a list of all previous operations. This functionality can be moved out of the calculator class, so that an external (let's call it undo manager class) can collect the internal state of the calculator and save it. However providing the explicit access to every state variable of the calculator to the restore manager would be impractical and would violate the encapsulation principle.

Intent

Type here to search

28°C ENG 18:27 03-08-2023

M2006C3MII

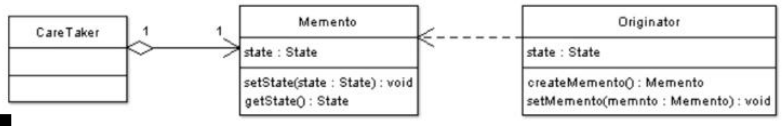
6:29 PM

Intent

- The intent of this pattern is to capture the internal state of an object without violating encapsulation and thus providing a mean for restoring the object into initial state when needed.

Implementation

The figure below shows a UML class diagram for the Memento Pattern:

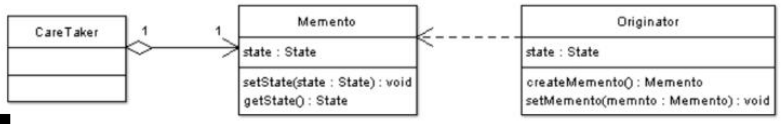


Type here to search

28°C ENG 18:27 03-08-2023

M2006C3MII

The figure below shows a UML class diagram for the Memento Pattern:



- **Memento**
  - Stores internal state of the Originator object. The state can include any number of state variables.
  - The Memento must have two interfaces, an interface to the caretaker. This interface must not allow any operations or any access to internal state stored by the memento and thus honors encapsulation. The other interface is to the originator

Type here to search 28°C ENG 18:27 03-08-2023

M2006C3MII

and allows the originator to access any state variables necessary to for the originator to restore previous state.

- **Originator**
  - Creates a memento object capturing the originators internal state.
  - Use the memento object to restore its previous state.
- **Caretaker**
  - Responsible for keeping the memento.
  - The memento is opaque to the caretaker, and the caretaker must not operate on it.

A Caretaker would like to perform an operation on the Originator while having the possibility to rollback. The caretaker calls the createMemento() method on the originator asking the originator to pass it a memento object. At this point the originator creates a memento object saving its internal state and passes the memento to the caretaker. The caretaker maintains the memento object and performs the operation. In case of the need to undo the operation, the caretaker calls the setMemento() method on the originator passing the maintained memento object. The originator would accept the memento, using it to restore its previous state.

Type here to search 28°C ENG 18:27 03-08-2023

M2006C3MII

6:29 PM

Applicability & Examples

The memento pattern is used when a snapshot of an object's state must be captured so that it can be restored to that state later and in situations where explicitly passing the state of the object would violate encapsulation.

Memento pattern is used to reduce where we want to restore state of an object to a previous state.

Memento pattern falls under behavioral pattern category.

Memento pattern uses three actor classes. Memento contains state of an object to be restored. Inventor creates and stores states in Memento objects and Warden object which is responsible to restore object state from Memento.

In this example we've created classes *Memento*, *Inventor* and *Warden*. *MementoPatternDemo* demo class will use *Warden* and *Inventor* objects to show restoration of object states.

Class Diagram:

Type here to search

28°C ENG 18:27 03-08-2023

M2006C3MII

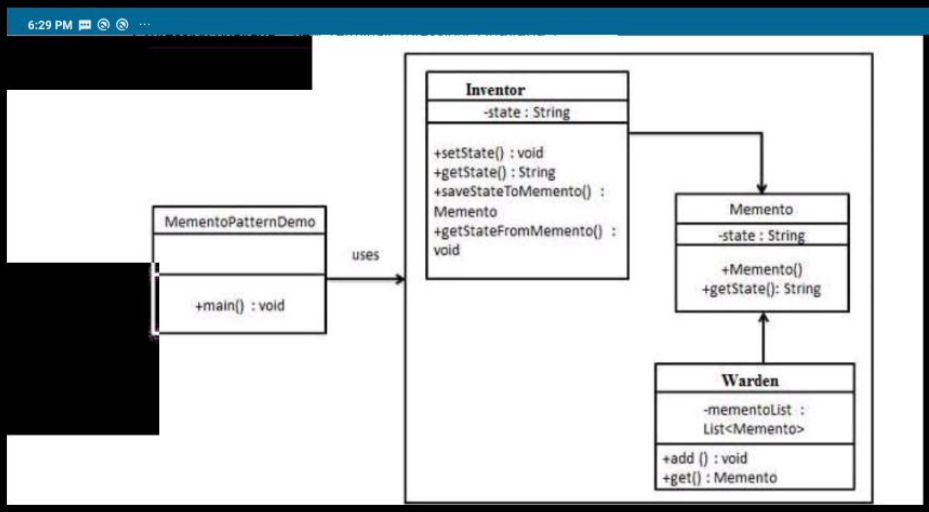
6:29 PM

Class Diagram:

Type here to search

28°C ENG 18:27 03-08-2023

M2006C3MII



Type here to search 28°C ENG 18:27 03-08-2023

M2006C3MII

Steps

Use the following steps to implement the above mentioned design pattern.

Step 1

Create Memento class.

Memento.java

```
public class Memento {
private String state;
public Memento(String state){
this.state = state;
}
public String getState(){
return state;
}
}
```

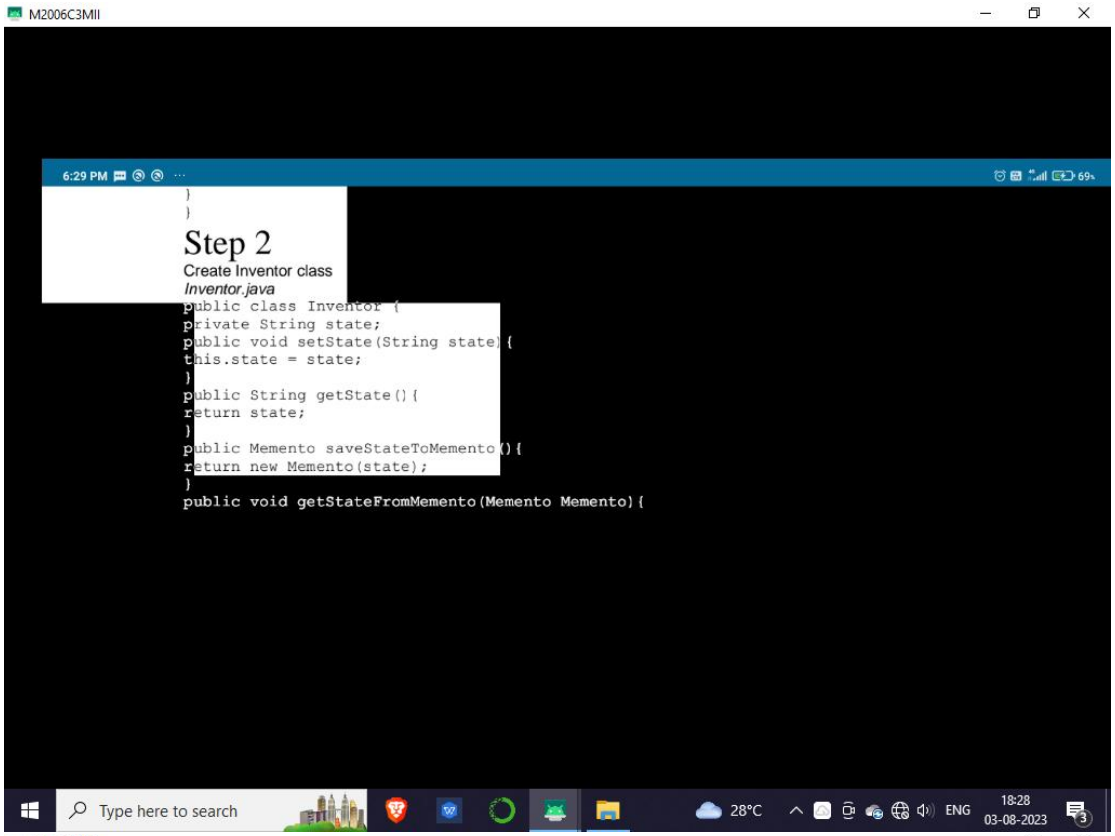
Step 2

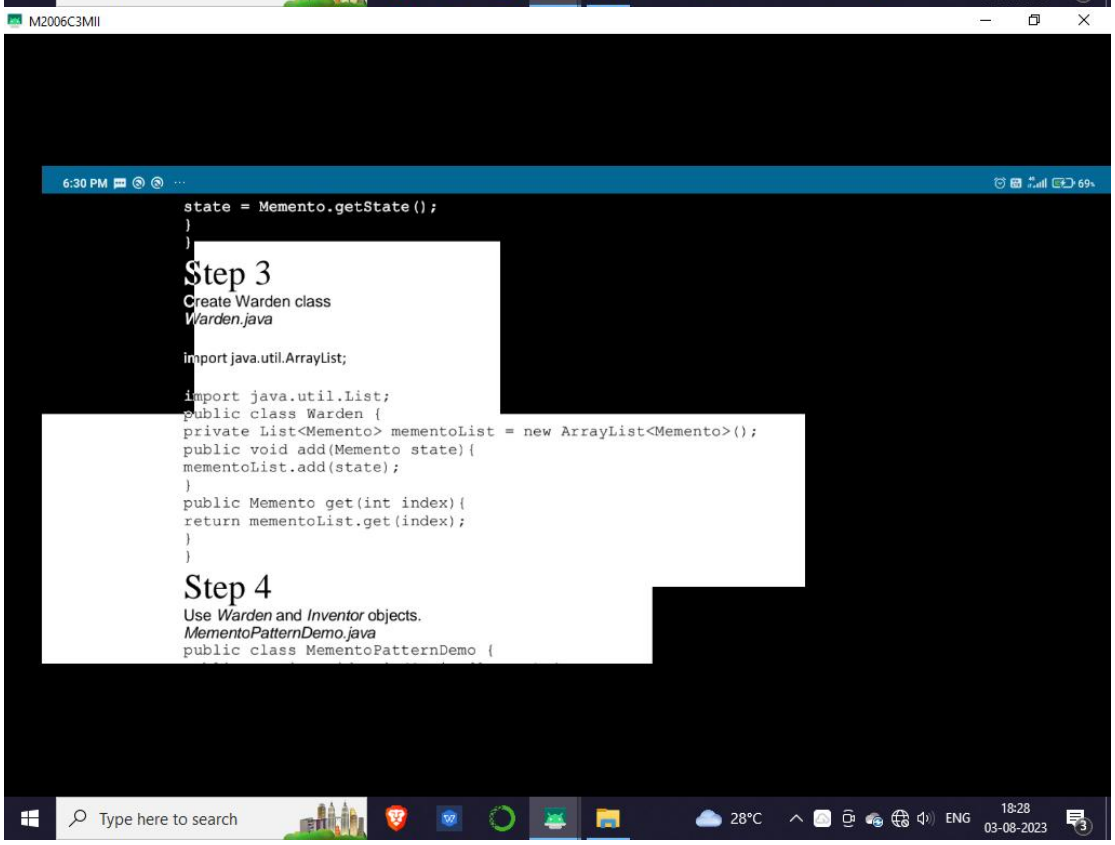
Create Inventor class

Inventor.java

```
public class Inventor {
private String state;
public void setState(String state){
this.state = state;
}
```

Type here to search 28°C ENG 18:28 03-08-2023


M2006C3MII
6:29 PM
}
}
Step 2
Create Inventor class
Inventor.java
public class Inventor {
private String state;
public void setState(String state){
this.state = state;
}
public String getState(){
return state;
}
public Memento saveStateToMemento(){
return new Memento(state);
}
public void getStateFromMemento(Memento Memento){
Type here to search
28°C
ENG
18:28
03-08-2023


M2006C3MII
6:30 PM
state = Memento.getState();
}
}
Step 3
Create Warden class
Warden.java
import java.util.ArrayList;
import java.util.List;
public class Warden {
private List<Memento> mementoList = new ArrayList<Memento>();
public void add(Memento state){
mementoList.add(state);
}
public Memento get(int index){
return mementoList.get(index);
}
}
Step 4
Use Warden and Inventor objects.
MementoPatternDemo.java
public class MementoPatternDemo {
Type here to search
28°C
ENG
18:28
03-08-2023

M2006C3MII

Step 4

Use *Warden* and *Inventor* objects.

MementoPatternDemo.java

```
public class MementoPatternDemo {
public static void main(String[] args) {
Inventor inventor = new Inventor();
Warden warden = new Warden();
inventor.setState("State #1");
inventor.setState("State #2");
warden.add(originator.saveStateToMemento());
inventor.setState("State #3");
warden.add(originator.saveStateToMemento());
inventor.setState("State #4");
System.out.println("Current State: " + inventor.getState());
inventor.getStateFromMemento(warden.get(0));
System.out.println("First saved State: " + inventor.getState());
inventor.getStateFromMemento(warden.get(1));
System.out.println("Second saved State: " + inventor.getState());
}
}
```

Step 5

output.

```
Current State: State #4
```

M2006C3MII

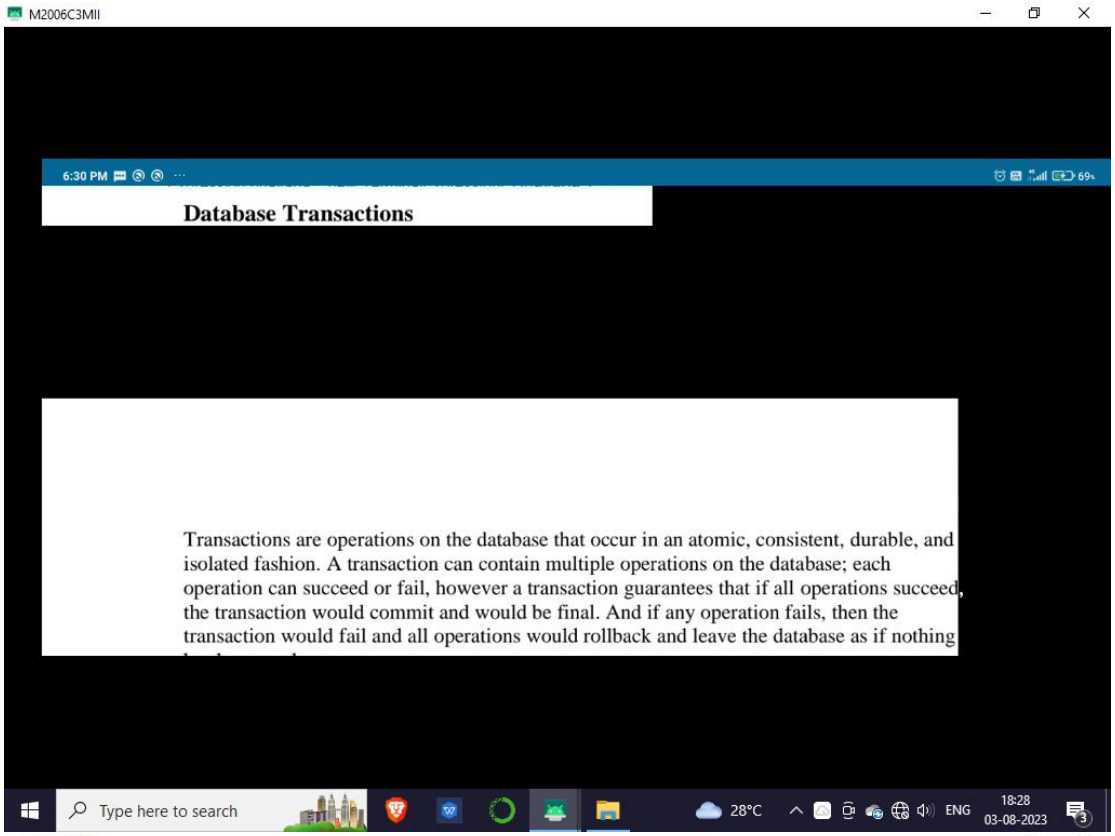
Step 5

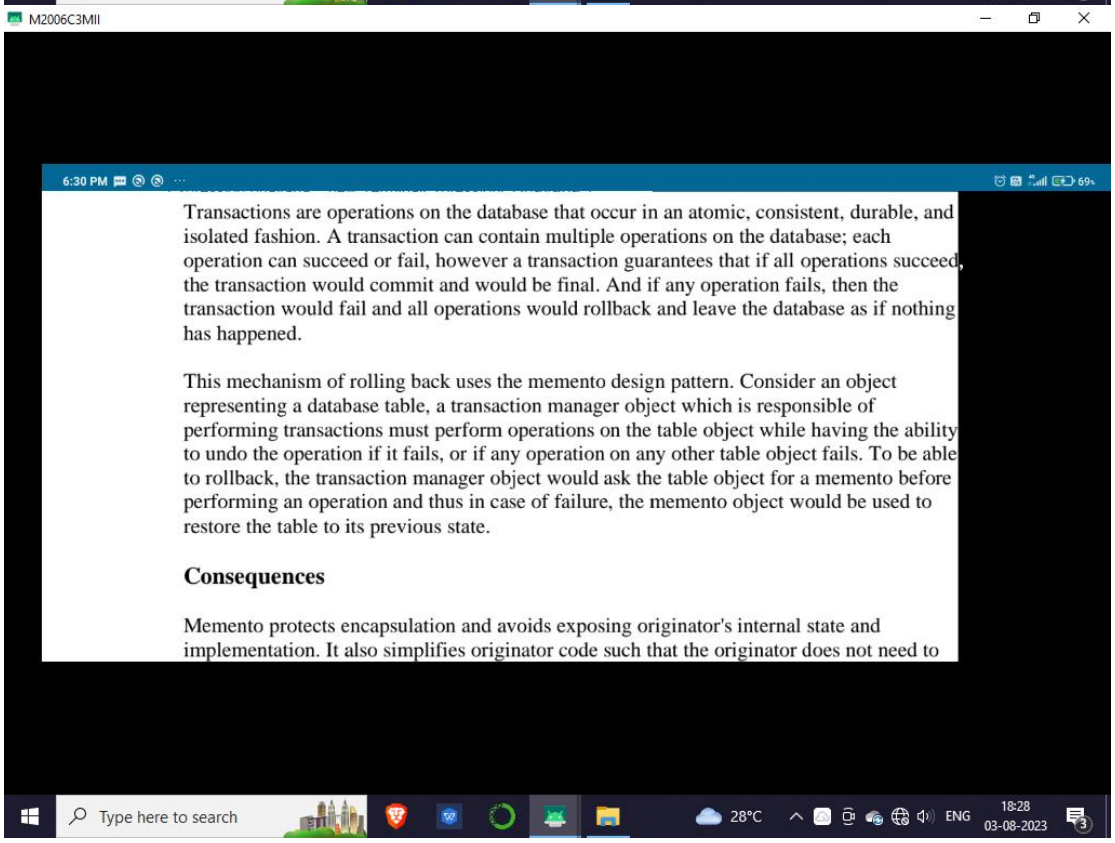
output.

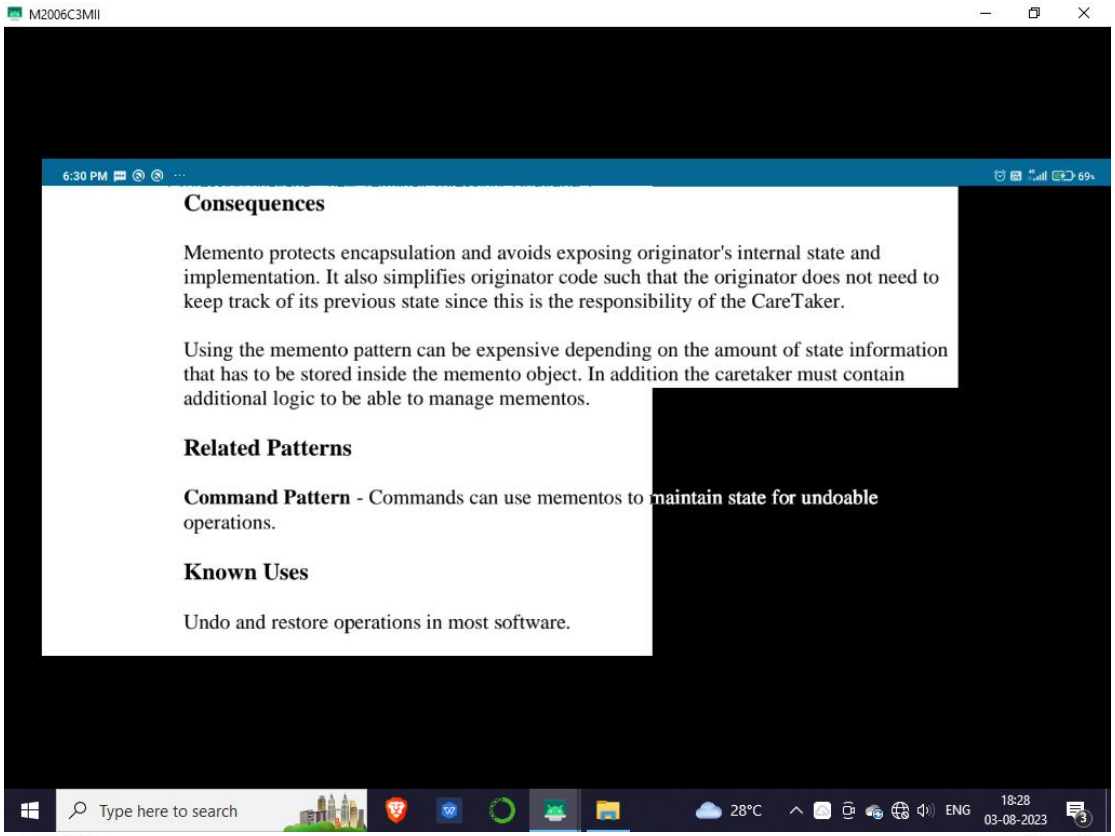
```
Current State: State #4
First saved State: State #2
Second saved State: State #3
```

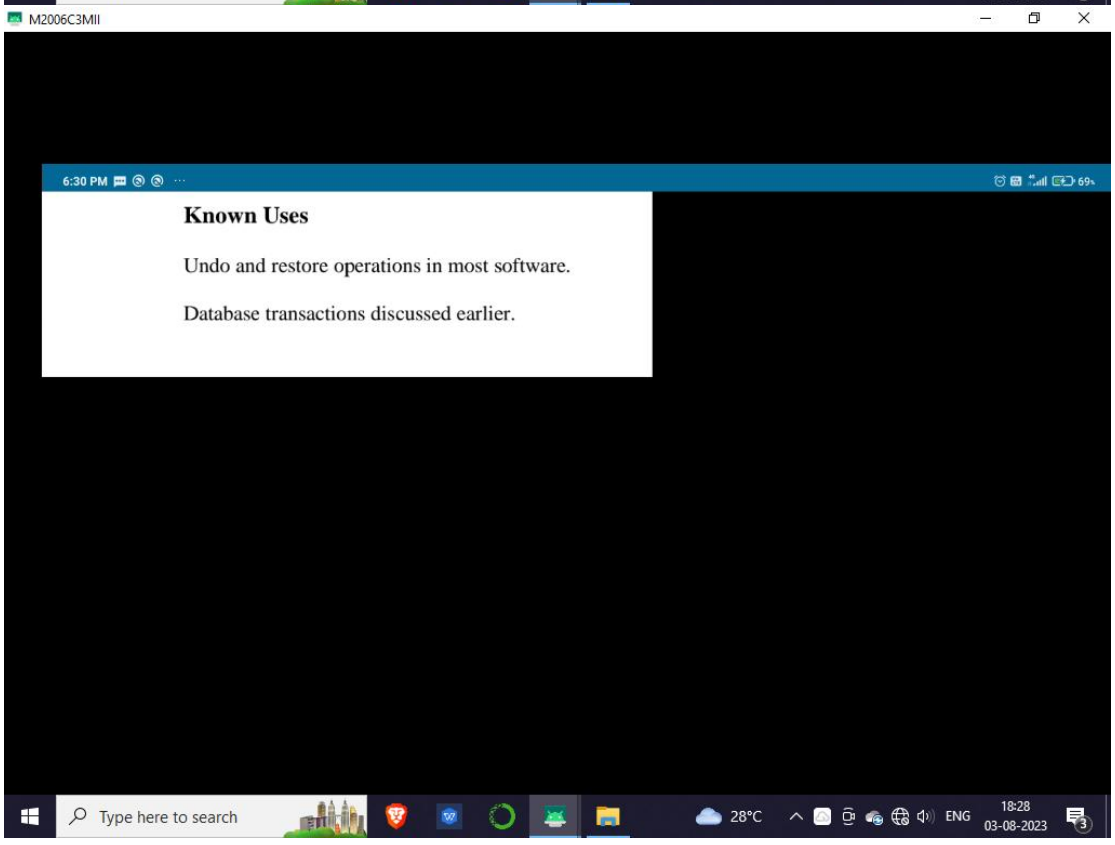
Specific problems and implementation

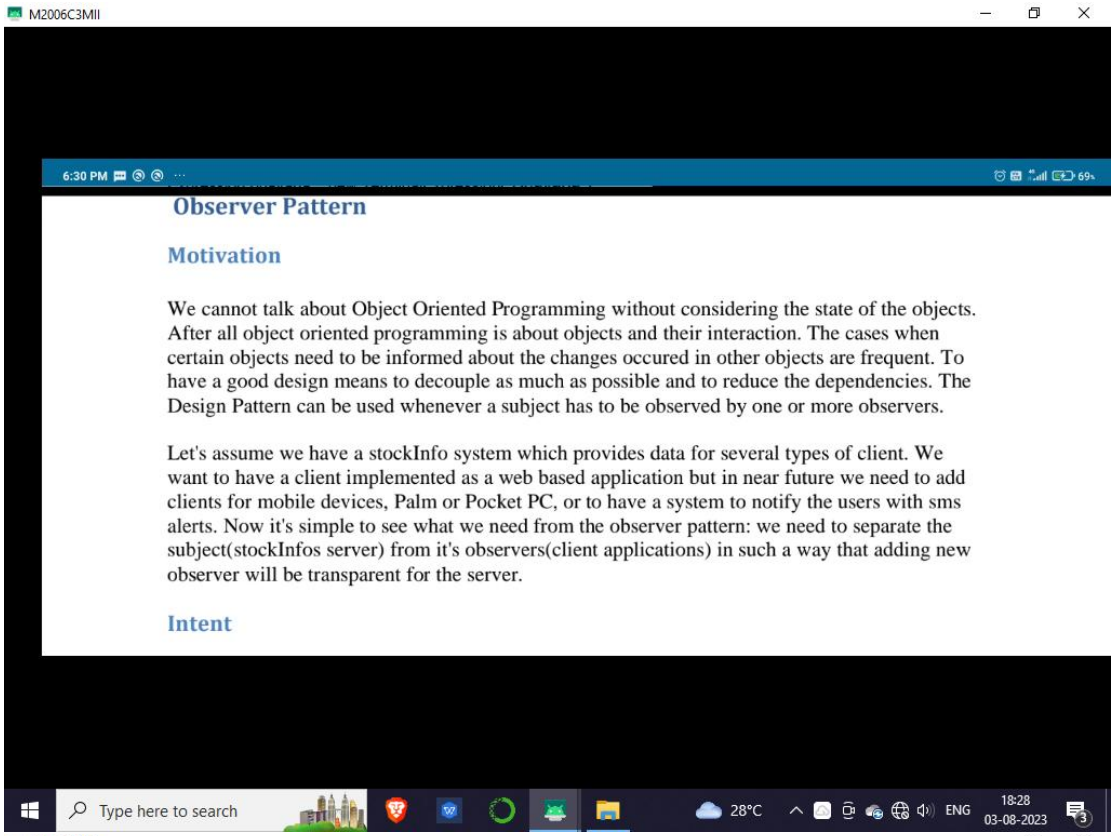
Database Transactions

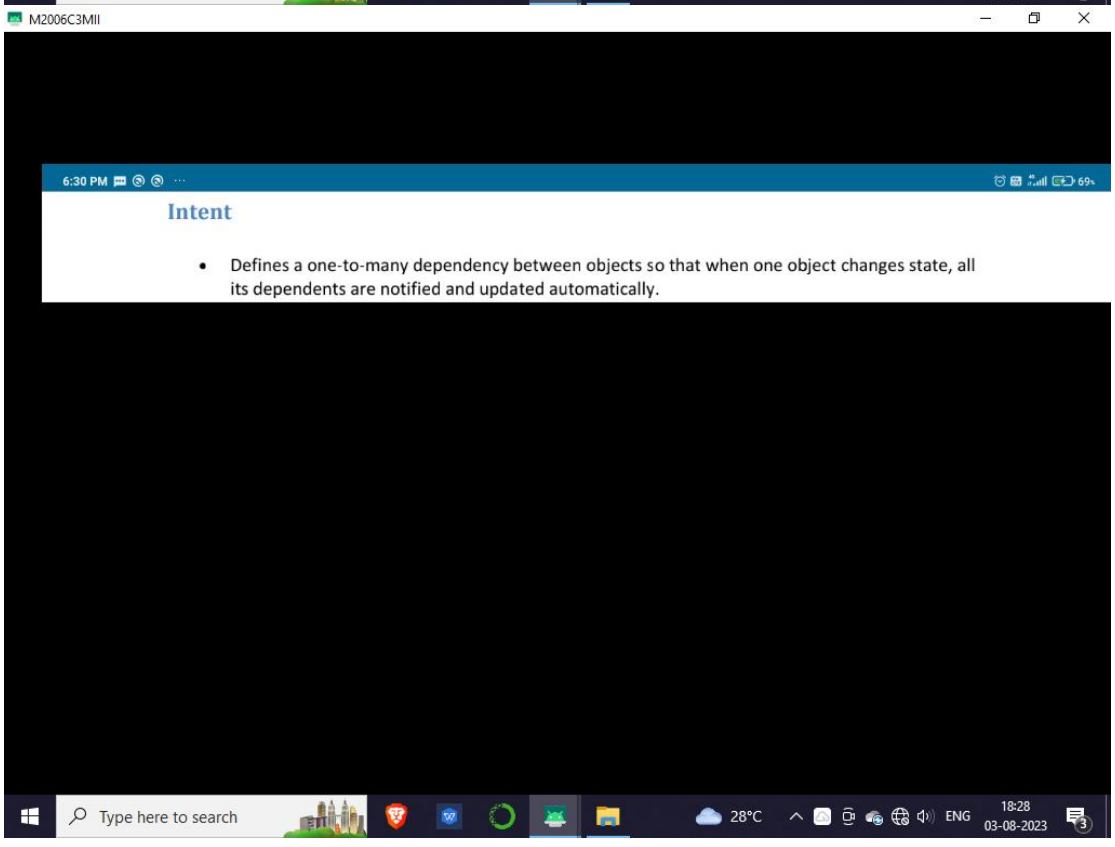

M2006C3MII
6:30 PM
Database Transactions
Transactions are operations on the database that occur in an atomic, consistent, durable, and isolated fashion. A transaction can contain multiple operations on the database; each operation can succeed or fail, however a transaction guarantees that if all operations succeed, the transaction would commit and would be final. And if any operation fails, then the transaction would fail and all operations would rollback and leave the database as if nothing
Type here to search
28°C
ENG
18:28
03-08-2023

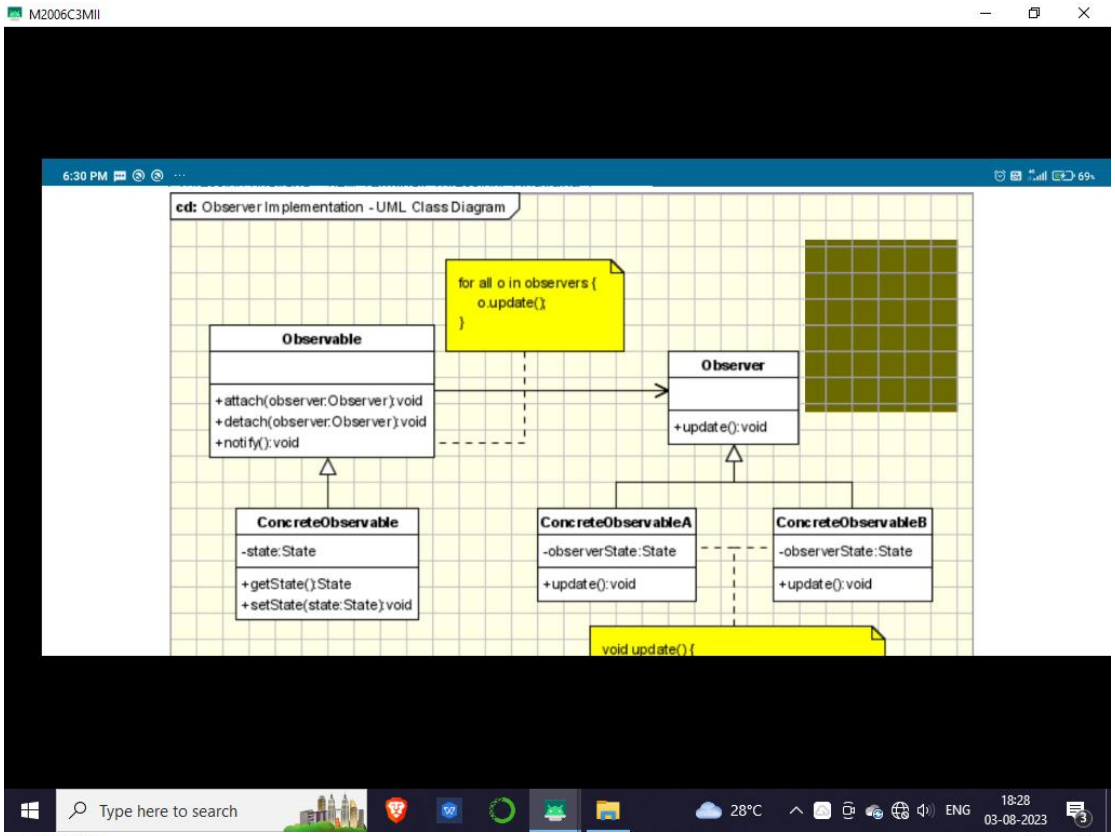

M2006C3MII
6:30 PM
Transactions are operations on the database that occur in an atomic, consistent, durable, and isolated fashion. A transaction can contain multiple operations on the database; each operation can succeed or fail, however a transaction guarantees that if all operations succeed, the transaction would commit and would be final. And if any operation fails, then the transaction would fail and all operations would rollback and leave the database as if nothing has happened.
This mechanism of rolling back uses the memento design pattern. Consider an object representing a database table, a transaction manager object which is responsible of performing transactions must perform operations on the table object while having the ability to undo the operation if it fails, or if any operation on any other table object fails. To be able to rollback, the transaction manager object would ask the table object for a memento before performing an operation and thus in case of failure, the memento object would be used to restore the table to its previous state.
Consequences
Memento protects encapsulation and avoids exposing originator's internal state and implementation. It also simplifies originator code such that the originator does not need to
Type here to search
28°C
ENG
18:28
03-08-2023

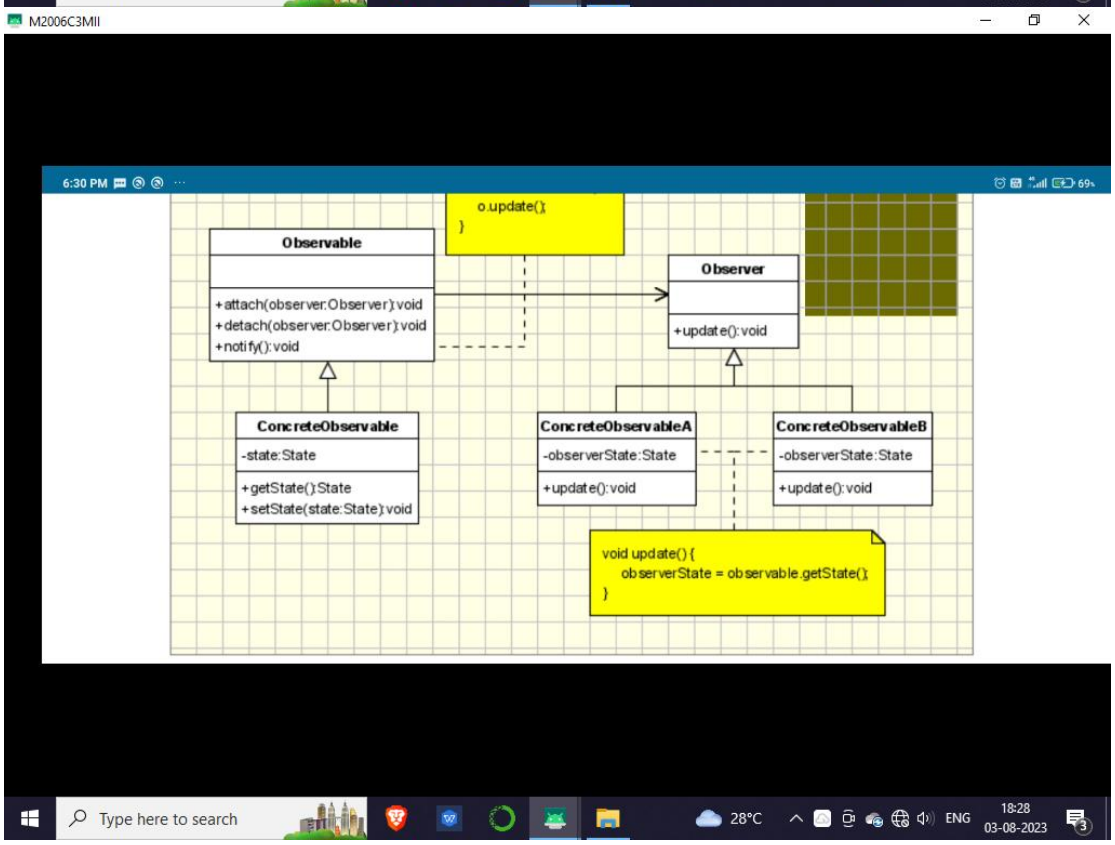

M2006C3MII
6:30 PM
Consequences
Memento protects encapsulation and avoids exposing originator's internal state and implementation. It also simplifies originator code such that the originator does not need to keep track of its previous state since this is the responsibility of the CareTaker.
Using the memento pattern can be expensive depending on the amount of state information that has to be stored inside the memento object. In addition the caretaker must contain additional logic to be able to manage mementos.
Related Patterns
Command Pattern - Commands can use mementos to maintain state for undoable operations.
Known Uses
Undo and restore operations in most software.
Type here to search
28°C
ENG
18:28
03-08-2023


M2006C3MII
6:30 PM
Known Uses
Undo and restore operations in most software.
Database transactions discussed earlier.
Type here to search
28°C
ENG
18:28
03-08-2023


M2006C3MII
6:30 PM
Observer Pattern
Motivation
We cannot talk about Object Oriented Programming without considering the state of the objects. After all object oriented programming is about objects and their interaction. The cases when certain objects need to be informed about the changes occured in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Design Pattern can be used whenever a subject has to be observed by one or more observers.
Let's assume we have a stockInfo system which provides data for several types of client. We want to have a client implemented as a web based application but in near future we need to add clients for mobile devices, Palm or Pocket PC, or to have a system to notify the users with sms alerts. Now it's simple to see what we need from the observer pattern: we need to separate the subject(stockInfos server) from it's observers(client applications) in such a way that adding new observer will be transparent for the server.
Intent
Type here to search
28°C
ENG
18:28
03-08-2023


M2006C3MII
6:30 PM
Intent
• Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Type here to search
28°C
ENG
18:28
03-08-2023


M2006C3MII
6:30 PM
cd: Observer Implementation - UML Class Diagram
for all o in observers {
o.update();
}
Observable
+attach(observer:Observer):void
+detach(observer:Observer):void
+notify():void
Observer
+update():void
ConcreteObservable
-state:State
+getState():State
+setState(state:State):void
ConcreteObservableA
-observerState:State
+update():void
ConcreteObservableB
-observerState:State
+update():void
void update() {
Type here to search
28°C
ENG
18:28
03-08-2023


M2006C3MII
6:30 PM
o.update();
}
Observable
+attach(observer:Observer):void
+detach(observer:Observer):void
+notify():void
Observer
+update():void
ConcreteObservable
-state:State
+getState():State
+setState(state:State):void
ConcreteObservableA
-observerState:State
+update():void
ConcreteObservableB
-observerState:State
+update():void
void update() {
observerState = observable.getState();
}
Type here to search
28°C
ENG
18:28
03-08-2023

M2006C3MII

6:30 PM

Implementation

The participants classes in this pattern are:

- **Observable** - interface or abstract class defining the operations for attaching and de-attaching observers to the client. In the GOF book this class/interface is known as **Subject**.
- **ConcreteObservable** - concrete Observable class. It maintain the state of the object and when a change in the state occurs it notifies the attached **Observers**.
- **Observer** - interface or abstract class defining the operations to be used to notify this object.
- **ConcreteObserverA, ConcreteObserver2** - concrete **Observer** implementations.

The flow is simple: the main framework instantiate the ConcreteObservable object. Then it instantiate and attaches the concrete observers to it using the methods defined in the Observable interface. Each time the state of the subject it's changing it notifies all the attached Observers using the methods defined in the Observer interface. When a new Observer is added to the application, all we need to do is to instantiate it in the main framework and to add attach it to the Observable object. The classes already created will remain unchanged.

Type here to search 28°C ENG 18:29 03-08-2023

M2006C3MII

6:31 PM

Applicability & Examples

The observer pattern is used when:

- the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
- the framework we are writing needs to be enhanced in future with new observers with minimal changes.

Some Classical Examples:

- **Model View Controller Pattern** - The observer pattern is used in the model view controller (MVC) architectural pattern. In MVC the this pattern is used to decouple the model from the view. View represents the **Observer** and the model is the **Observable** object.
- **Event management** - This is one of the domains where the Observer patterns is extensively used. Swing and .Net are extensively using the Observer pattern for implementing the events mechanism.

Type here to search 28°C ENG 18:29 03-08-2023

M2006C3MII

6:31 PM

Example:

Observer pattern is used when there is one to many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

Observer pattern uses three actor classes Subject, Observer and Client. Subject an object having methods to attach and de-attach observers to a client object. We've created classes *Subject*, *Observer*abstract class and concrete classes extending the abstract class the *Observer*.

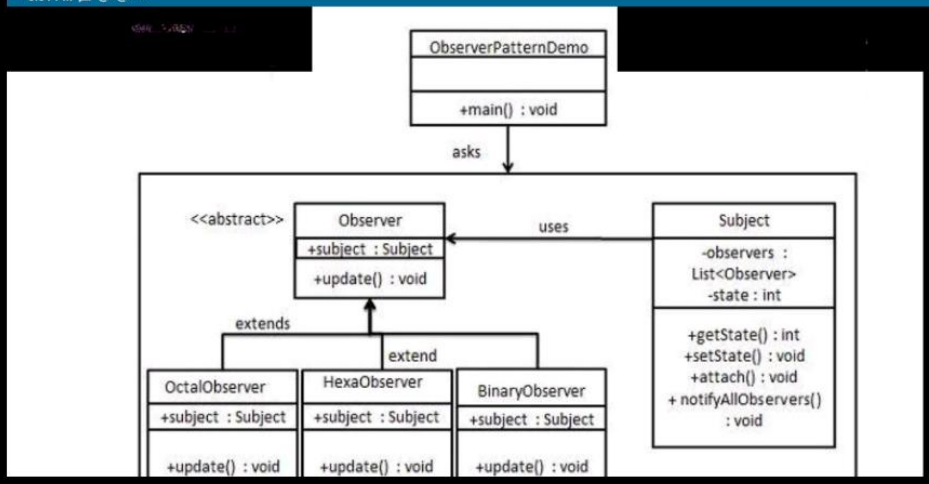
ObserverPatternDemo demo class will use *Subject* and concrete class objects to show observer pattern in action.

Type here to search

28°C ENG 18:29 03-08-2023

M2006C3MII

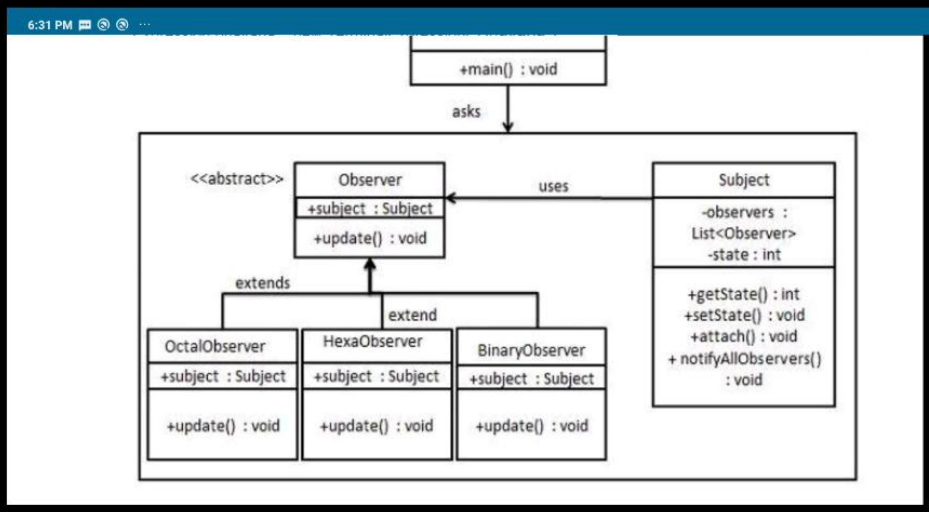
6:31 PM



Type here to search

28°C ENG 18:29 03-08-2023

M2006C3MII



M2006C3MII

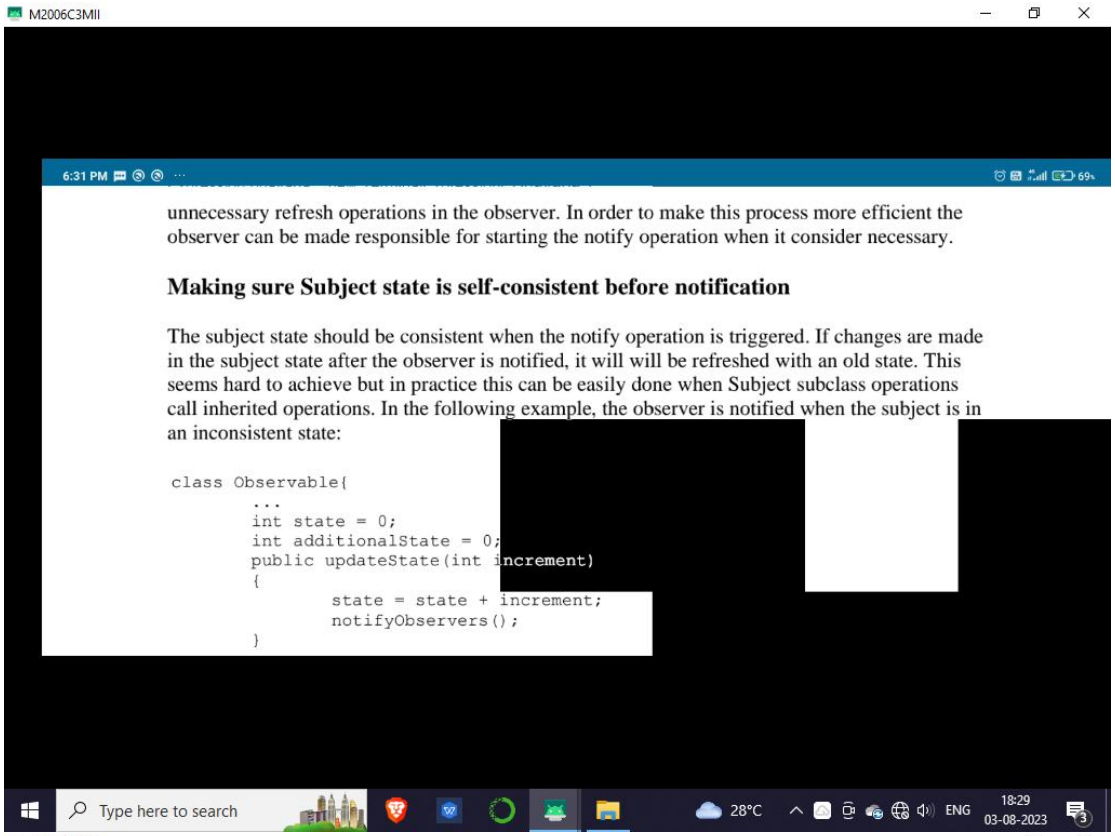
Specific Implementation Problems

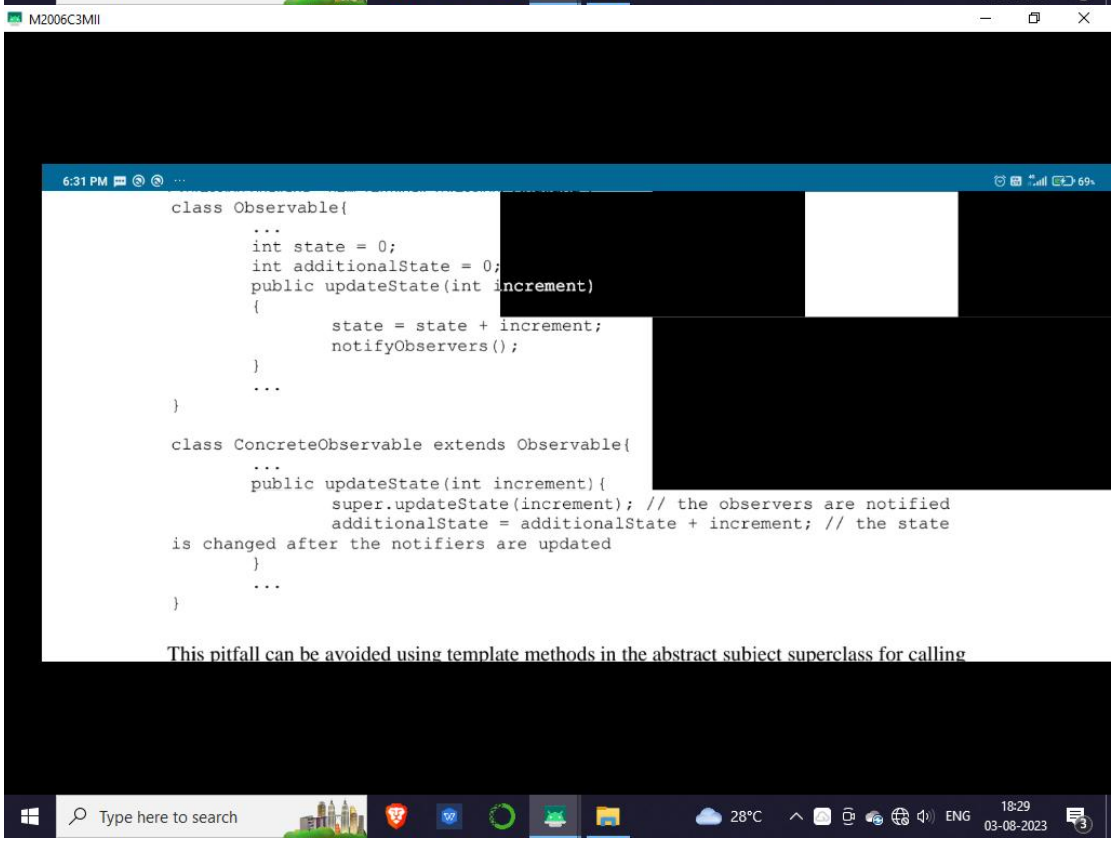
Many subjects to Many observers

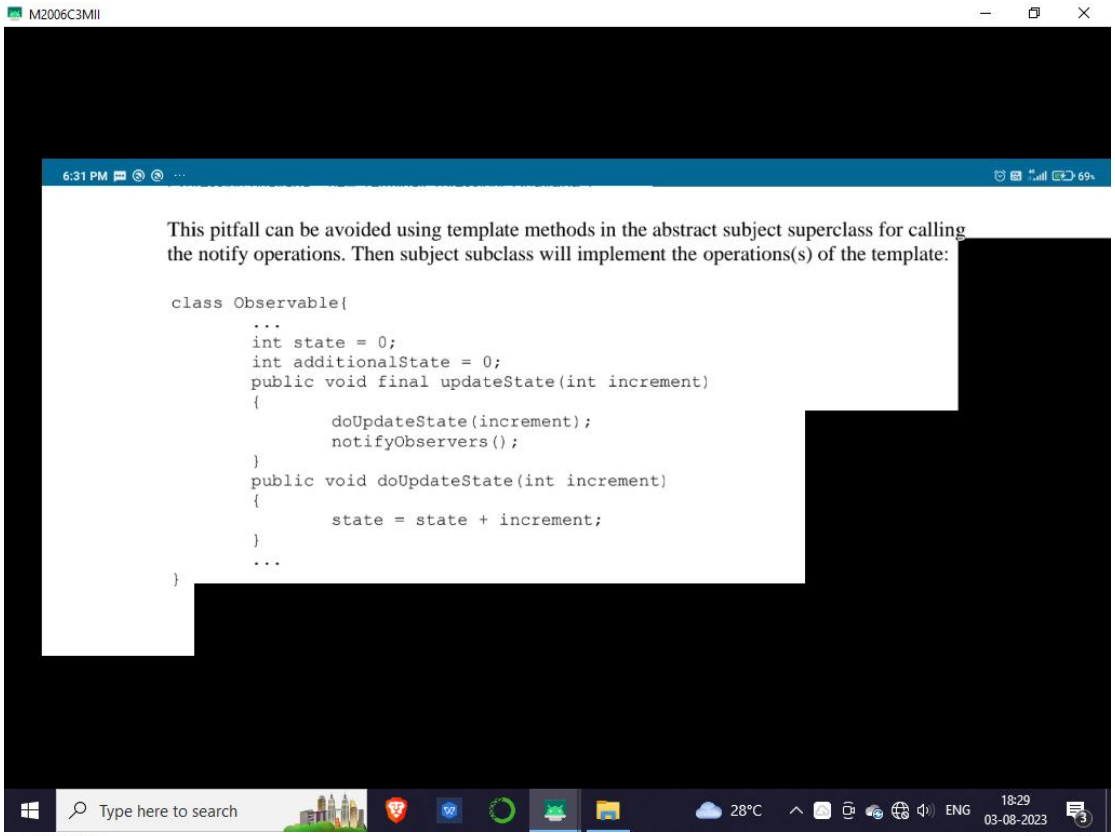
It's not a common situation but there are cases when a there are many observers that need to observe more than one subject. In this case the observer need to be notified not only about the change, but also which is the subject with the state changed. This can be realized very simple by adding to the subjects reference in the update notification method. The subject will pass a reference to itself(this) to the when notify the observer.

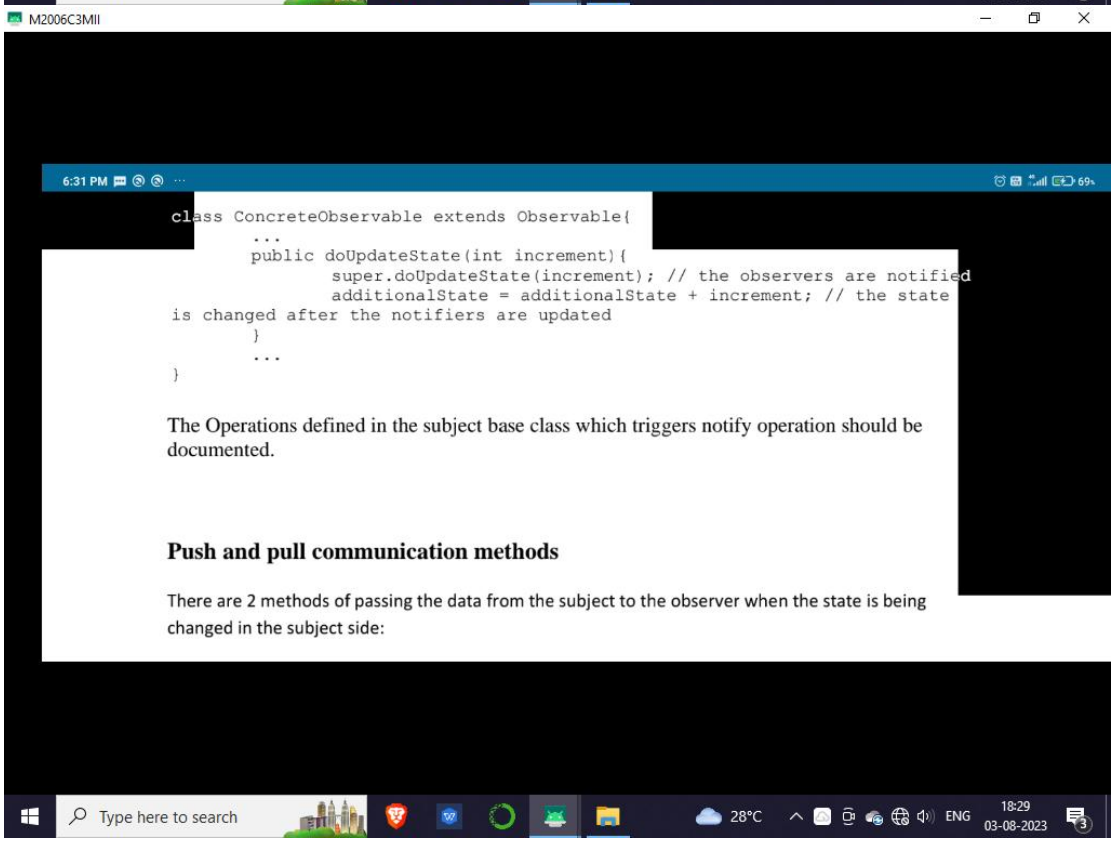
Who triggers the update?

The communication between the subject and its observers is done through the notify method declared in observer interface. But who it cat be triggered from either subject or observer object. Usually the notify method is triggered by the subject when it's state is changed. But sometimes when the updates are frequent the consecutive changes in the subject will determine many


M2006C3MII
6:31 PM
unnecessary refresh operations in the observer. In order to make this process more efficient the observer can be made responsible for starting the notify operation when it consider necessary.
Making sure Subject state is self-consistent before notification
The subject state should be consistent when the notify operation is triggered. If changes are made in the subject state after the observer is notified, it will will be refreshed with an old state. This seems hard to achieve but in practice this can be easily done when Subject subclass operations call inherited operations. In the following example, the observer is notified when the subject is in an inconsistent state:
class Observable{
...
int state = 0;
int additionalState = 0;
public updateState(int increment)
{
state = state + increment;
notifyObservers();
}
Type here to search
28°C
ENG
18:29
03-08-2023


M2006C3MII
6:31 PM
class Observable{
...
int state = 0;
int additionalState = 0;
public updateState(int increment)
{
state = state + increment;
notifyObservers();
}
...
}
class ConcreteObservable extends Observable{
...
public updateState(int increment){
super.updateState(increment); // the observers are notified
additionalState = additionalState + increment; // the state
is changed after the notifiers are updated
}
...
}
This pitfall can be avoided using template methods in the abstract subject superclass for calling
Type here to search
28°C
ENG
18:29
03-08-2023


M2006C3MII
6:31 PM
This pitfall can be avoided using template methods in the abstract subject superclass for calling the notify operations. Then subject subclass will implement the operations(s) of the template:
class Observable{
...
int state = 0;
int additionalState = 0;
public void final updateState(int increment)
{
doUpdateState(increment);
notifyObservers();
}
public void doUpdateState(int increment)
{
state = state + increment;
}
...
}
Type here to search
28°C
ENG
18:29
03-08-2023


M2006C3MII
6:31 PM
class ConcreteObservable extends Observable{
...
public doUpdateState(int increment){
super.doUpdateState(increment); // the observers are notified
additionalState = additionalState + increment; // the state
is changed after the notifiers are updated
}
...
}
The Operations defined in the subject base class which triggers notify operation should be documented.
Push and pull communication methods
There are 2 methods of passing the data from the subject to the observer when the state is being changed in the subject side:
Type here to search
28°C
ENG
18:29
03-08-2023

M2006C3MII

6:31 PM

Push and pull communication methods

There are 2 methods of passing the data from the subject to the observer when the state is being changed in the subject side:

- **Push model** - The subjects send detailed information about the change to the observer whether it uses it or not. Because the subject needs to send the detailed information to the observer this might be inefficient when a large amount of data needs to be sent and it is not used. Another aproach would be to send only the information required by the observer. In this case the subject should be able to distinguish between different types of observers and to know the required data of each of them, meaning that the subject layer is more coupled to observer layer.

- **Pull model** - The subject just notifies the observers when a change in his state appears and it's the responsibility of each observer to pull the required data from the subject. This can be inefficient because the communication is done in 2 steps and problems might appear in multithreading environments.

Specifying points of interests

Type here to search 28°C ENG 18:29 03-08-2023

M2006C3MII

6:31 PM

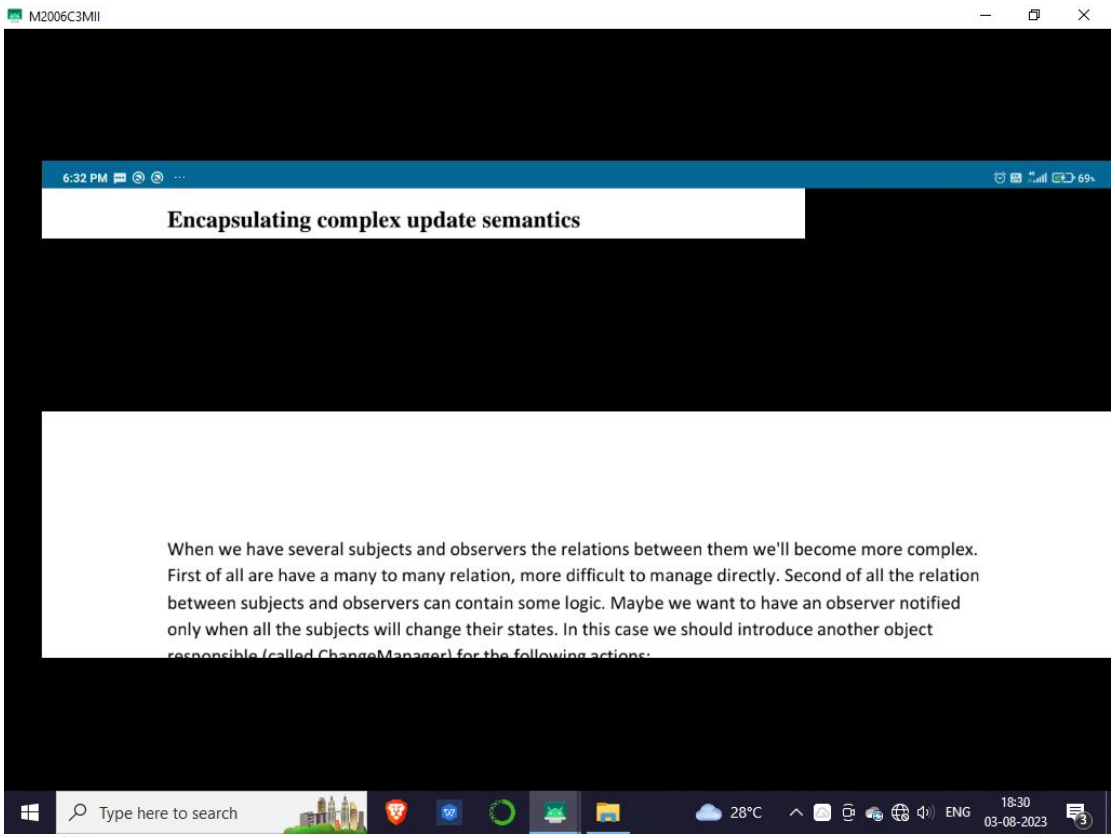
Specifying points of interests

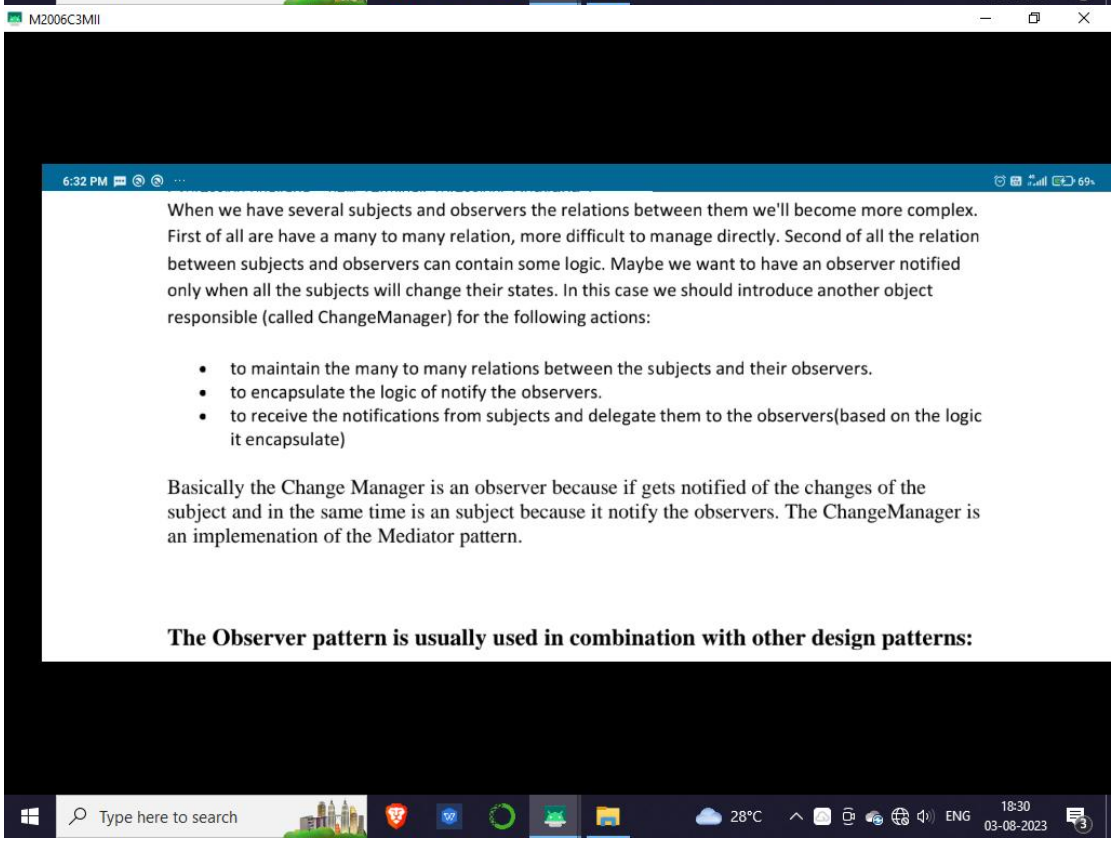
The efficiency can be improved by specifying which are the events on which each observer is interested. This can be realized by adding a new class defining an aspect. When an observer is registering it will provide the aspects in which it is interested:

```
class Subject{
        ...
        void attach(Observer observer, Aspect interest);
        ...
}
```

Encapsulating complex update semantics

Type here to search 28°C ENG 18:30 03-08-2023


M2006C3MII
6:32 PM
Encapsulating complex update semantics
When we have several subjects and observers the relations between them we'll become more complex. First of all are have a many to many relation, more difficult to manage directly. Second of all the relation between subjects and observers can contain some logic. Maybe we want to have an observer notified only when all the subjects will change their states. In this case we should introduce another object responsible (called ChangeManager) for the following actions:
Type here to search
28°C
ENG
18:30
03-08-2023


M2006C3MII
6:32 PM
When we have several subjects and observers the relations between them we'll become more complex. First of all are have a many to many relation, more difficult to manage directly. Second of all the relation between subjects and observers can contain some logic. Maybe we want to have an observer notified only when all the subjects will change their states. In this case we should introduce another object responsible (called ChangeManager) for the following actions:
to maintain the many to many relations between the subjects and their observers.
to encapsulate the logic of notify the observers.
to receive the notifications from subjects and delegate them to the observers(based on the logic it encapsulate)
Basically the Change Manager is an observer because if gets notified of the changes of the subject and in the same time is an subject because it notify the observers. The ChangeManager is an implemenation of the Mediator pattern.
The Observer pattern is usually used in combination with other design patterns:
Type here to search
28°C
ENG
18:30
03-08-2023

M2006C3MII

6:32 PM

The Observer pattern is usually used in combination with other design patterns:

- **Factory pattern** - It's very likely to use the factory pattern to create the Observers so no changes will be required even in the main framework. The new observers can be added directly in the configuration files.
- **Template Method** - The observer pattern can be used in conjunction with the Template Method Pattern to make sure that Subject state is self-consistent before notification
- **Mediator Pattern** - The mediator pattern can be used when we have cases of complex cases of many subjects an many observers

Strategy

Motivation

There are common situations when classes differ only in their behavior. For this cases is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at

Type here to search 28°C ENG 18:30 03-08-2023

M2006C3MII

6:32 PM

Strategy

Motivation

There are common situations when classes differ only in their behavior. For this cases is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime.

Intent

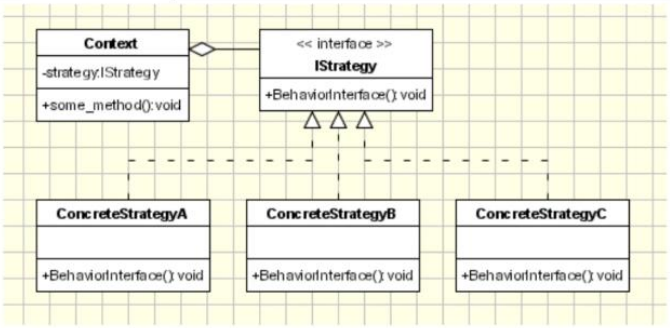
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Type here to search 28°C ENG 18:30 03-08-2023

M2006C3MII

6:32 PM

Implementation



Strategy - defines an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

Type here to search 28°C ENG 18:30 03-08-2023

M2006C3MII

6:32 PM

Strategy - defines an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy - each concrete strategy implements an algorithm.

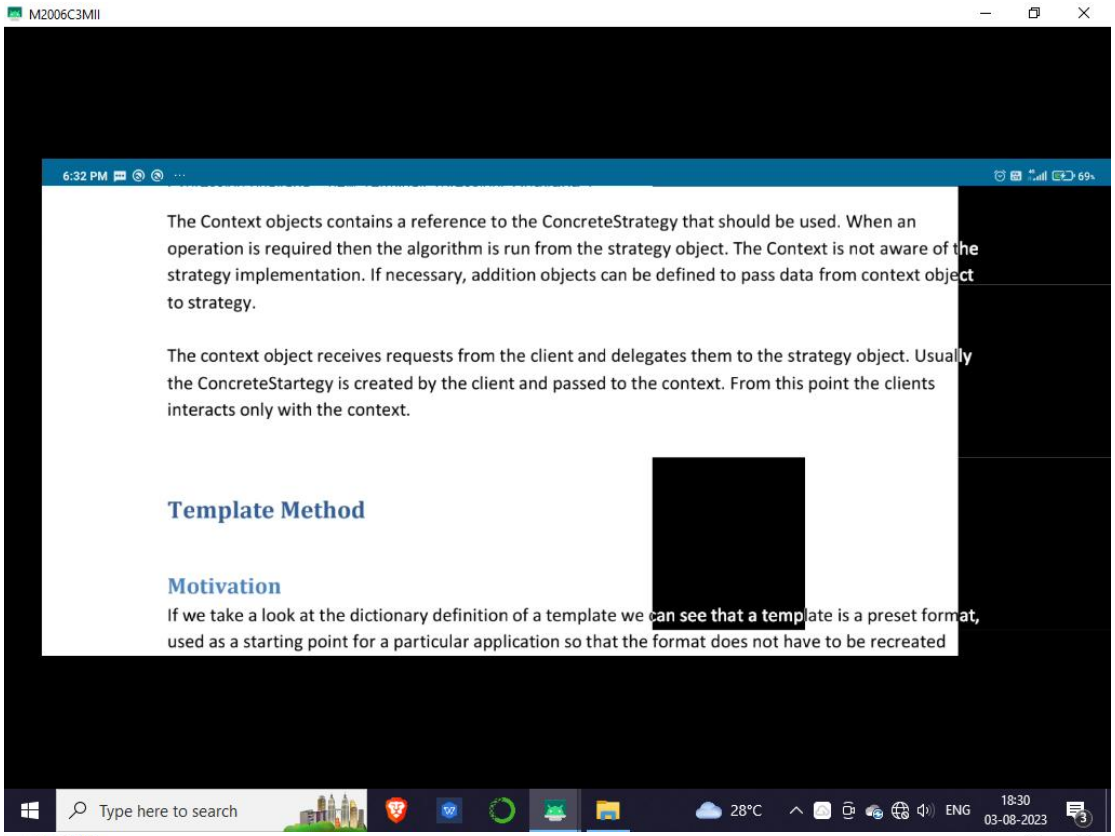
Context

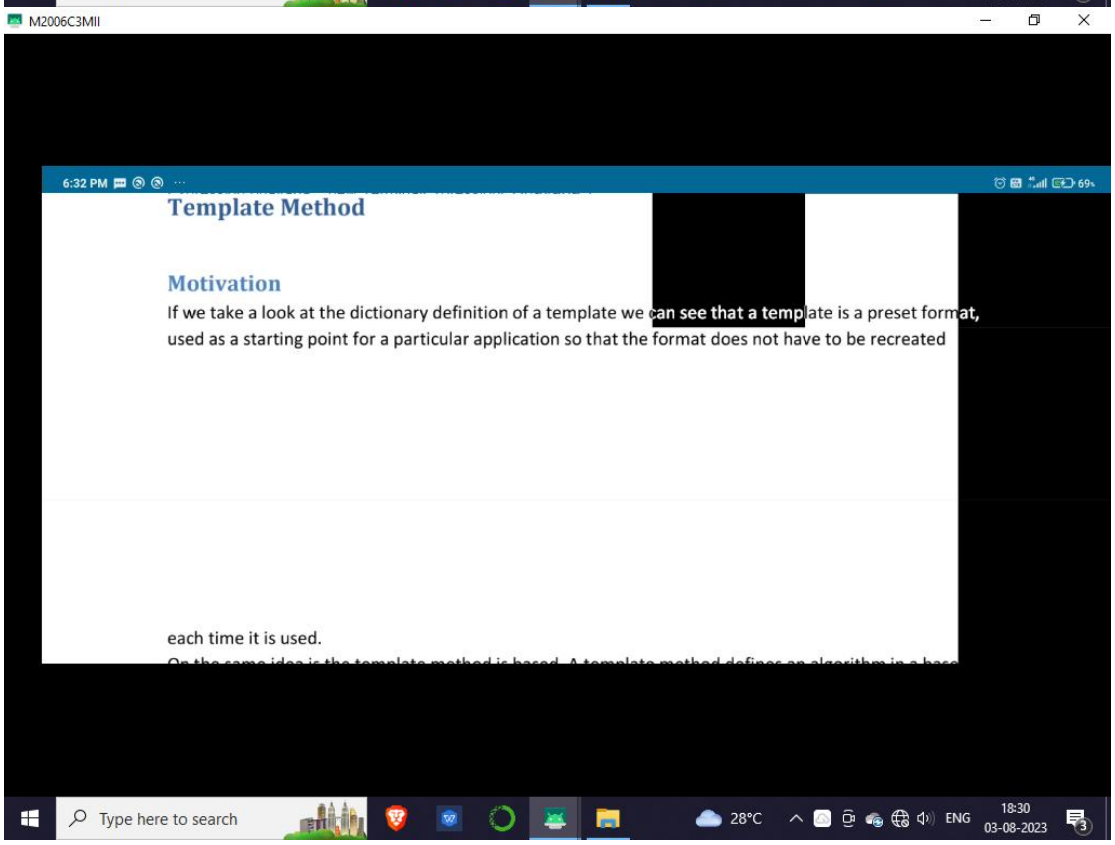
- contains a reference to a strategy object.
- may define an interface that lets strategy accessing its data.

The Context objects contains a reference to the ConcreteStrategy that should be used. When an operation is required then the algorithm is run from the strategy object. The Context is not aware of the strategy implementation. If necessary, addition objects can be defined to pass data from context object to strategy.

The context object receives requests from the client and delegates them to the strategy object. Usually the ConcreteStartegy is created by the client and passed to the context. From this point the clients interacts only with the context

Type here to search 28°C ENG 18:30 03-08-2023


M2006C3MII
6:32 PM
The Context objects contains a reference to the ConcreteStrategy that should be used. When an operation is required then the algorithm is run from the strategy object. The Context is not aware of the strategy implementation. If necessary, addition objects can be defined to pass data from context object to strategy.
The context object receives requests from the client and delegates them to the strategy object. Usually the ConcreteStartegy is created by the client and passed to the context. From this point the clients interacts only with the context.
Template Method
Motivation
If we take a look at the dictionary definition of a template we can see that a template is a preset format, used as a starting point for a particular application so that the format does not have to be recreated
Type here to search
28°C
ENG
18:30
03-08-2023


M2006C3MII
6:32 PM
Template Method
Motivation
If we take a look at the dictionary definition of a template we can see that a template is a preset format, used as a starting point for a particular application so that the format does not have to be recreated
each time it is used.
Type here to search
28°C
ENG
18:30
03-08-2023

M2006C3MII

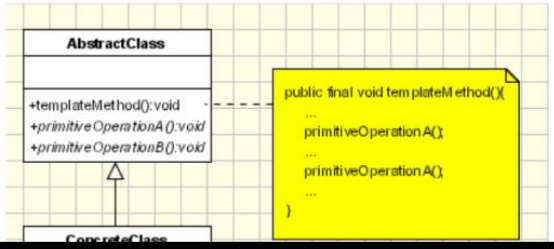
6:32 PM

each time it is used.

On the same idea is the template method is based. A template method defines an algorithm in a base class using abstract operations that subclasses override to provide concrete behavior.

Intent

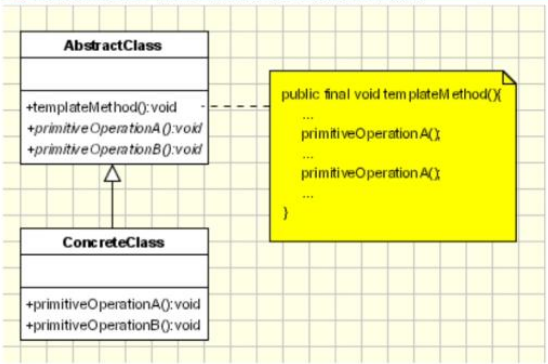
- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure.



Type here to search 28°C ENG 18:30 03-08-2023

M2006C3MII

6:32 PM



Implementation

AbstractClass - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.

Type here to search 28°C ENG 18:30 03-08-2023

M2006C3MII

6:32 PM

Implementation

AbstractClass - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.

- implements a template method which defines the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.

ConcreteClass - implements the primitive operations to carry out subclass-specific steps of the algorithm.

When a concrete class is called the template method code will be executed from the base class while for each method used inside the template method will be called the implementation from the derived class.

Applicability & Examples

The Template Method pattern should be used:

- to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- when refactoring is performed and common behavior is identified among classes. A abstract base class containing all the common code (in the template method) should be created to avoid code duplication.

Type here to search

28°C ENG 18:30 03-08-2023

M2006C3MII

6:32 PM

Applicability & Examples

The Template Method pattern should be used:

- to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- when refactoring is performed and common behavior is identified among classes. A abstract base class containing all the common code (in the template method) should be created to avoid code duplication.

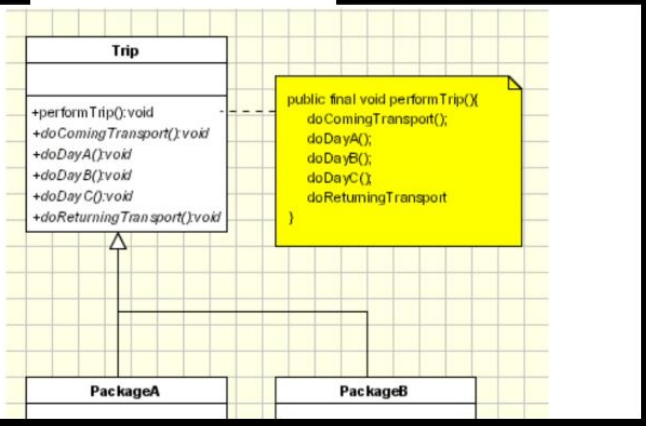
Type here to search

28°C ENG 18:30 03-08-2023

M2006C3MII

6:32 PM

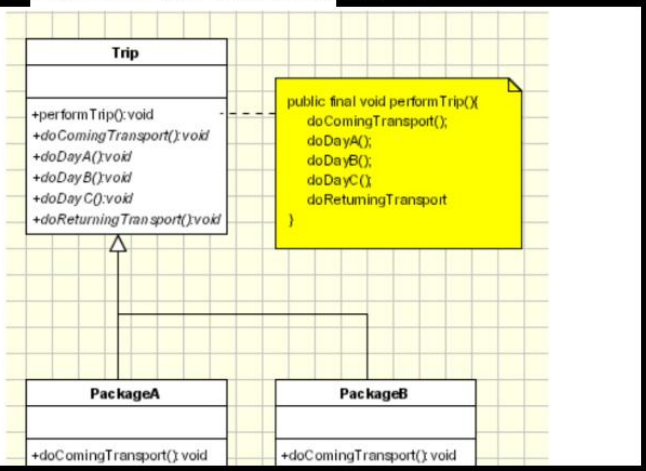
Example - Application used by a travel agency.



Type here to search 28°C ENG 18:30 03-08-2023

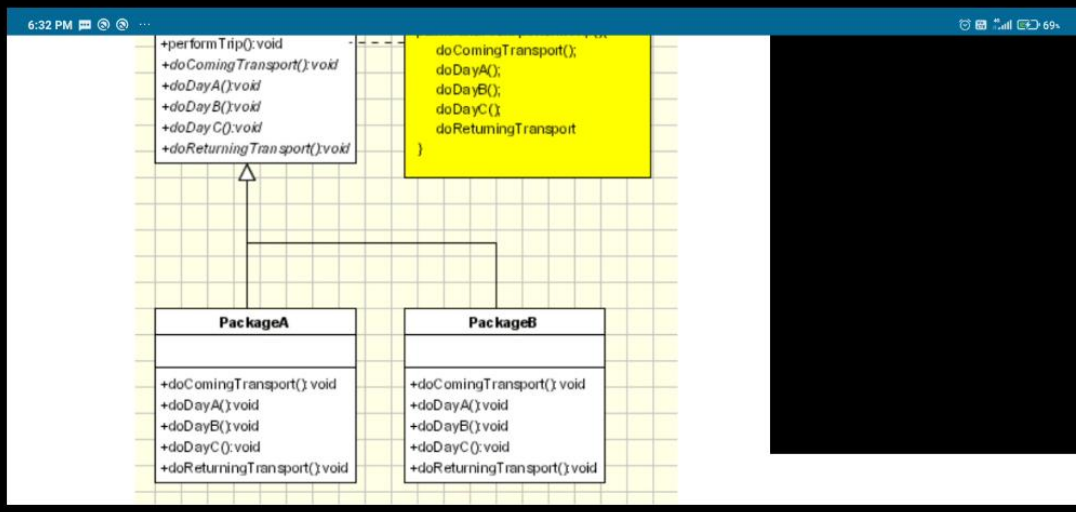
M2006C3MII

6:32 PM



Type here to search 28°C ENG 18:30 03-08-2023

M2006C3MII



M2006C3MII

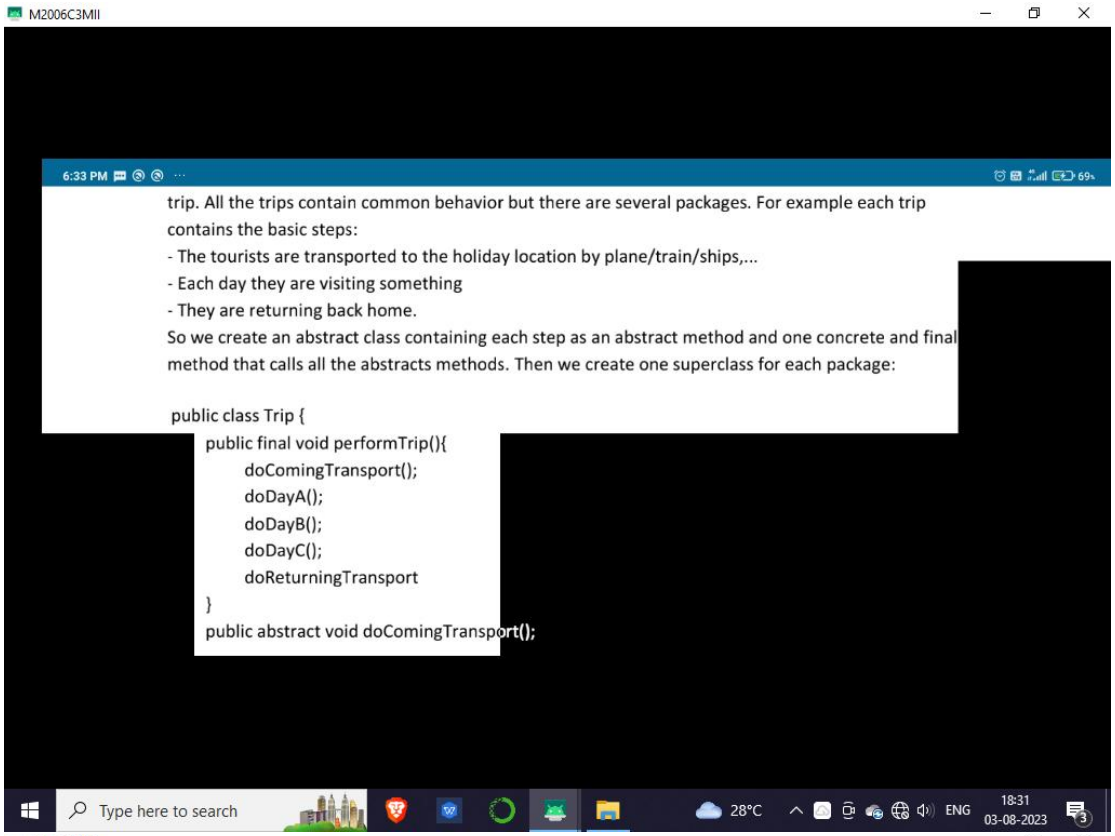
6:33 PM

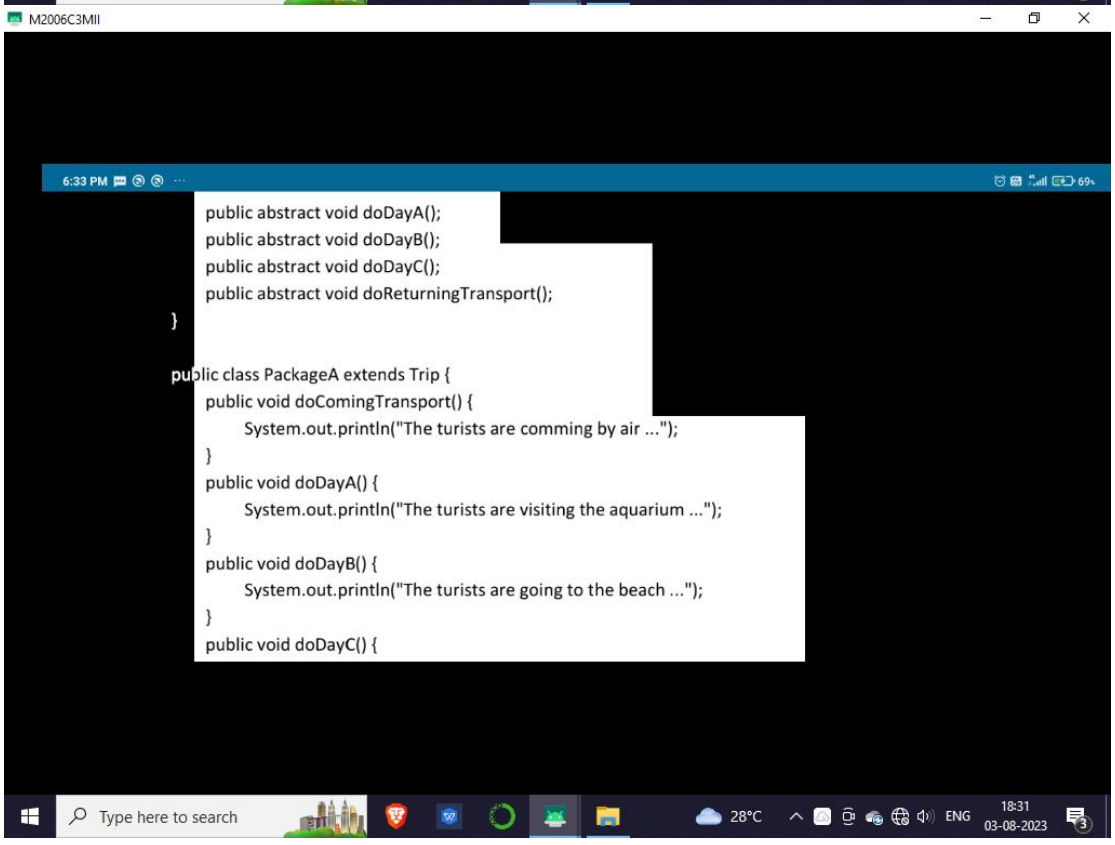
Lets' assume we have to develop an application for a travel agency. The travel agency is managing each trip. All the trips contain common behavior but there are several packages. For example each trip contains the basic steps:

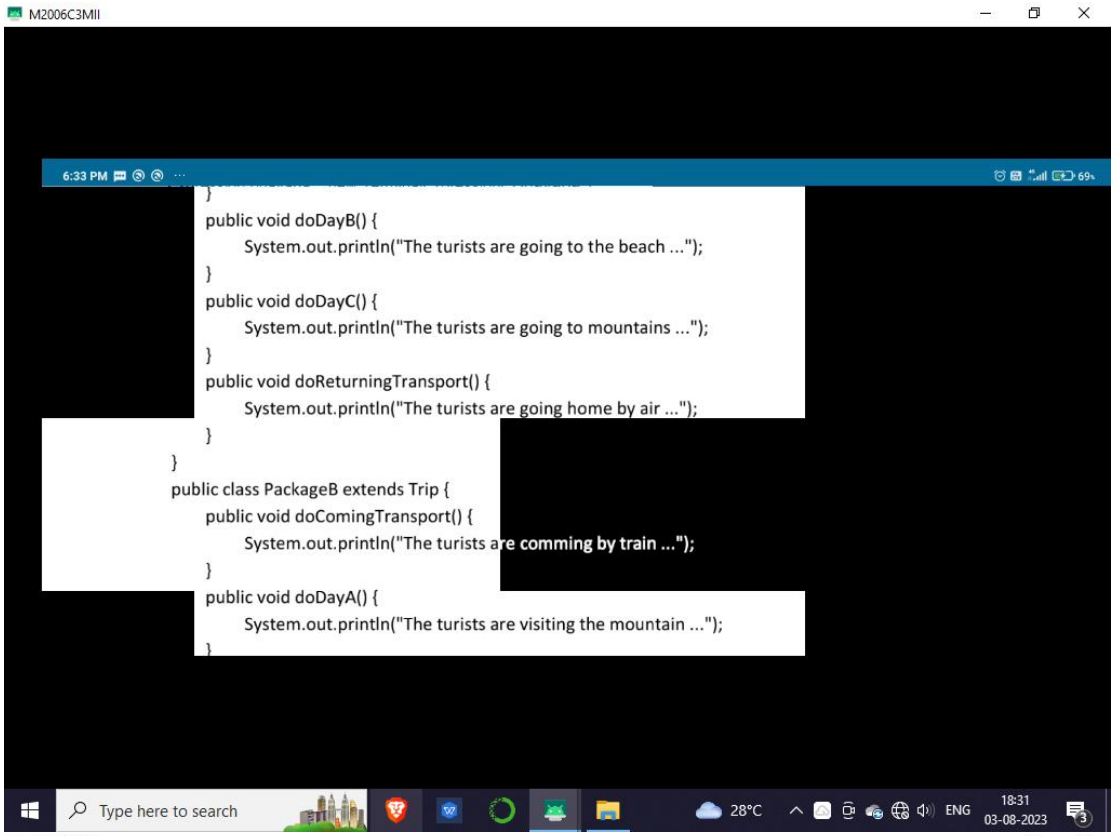
- The tourists are transported to the holiday location by plane/train/ships,...
- Each day they are visiting something
- They are returning back home.

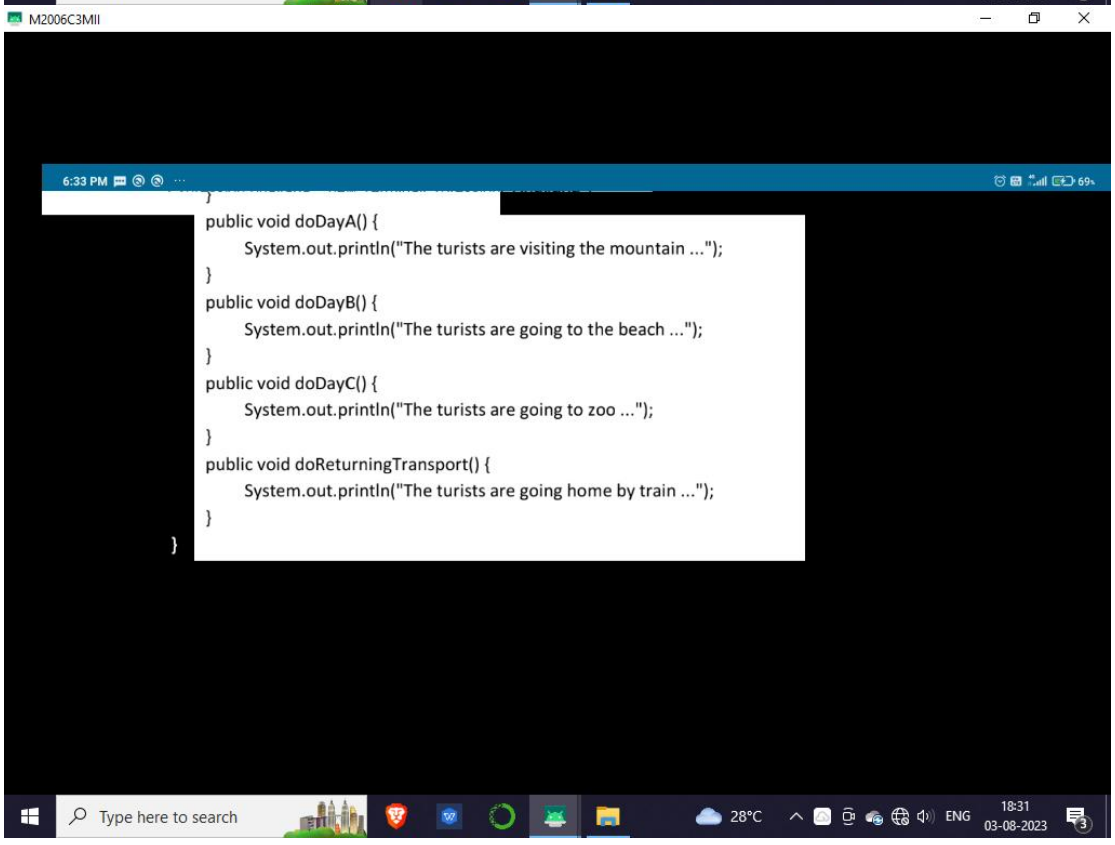
So we create an abstract class containing each step as an abstract method and one concrete and final method that calls all the abstracts methods. Then we create one superclass for each package:

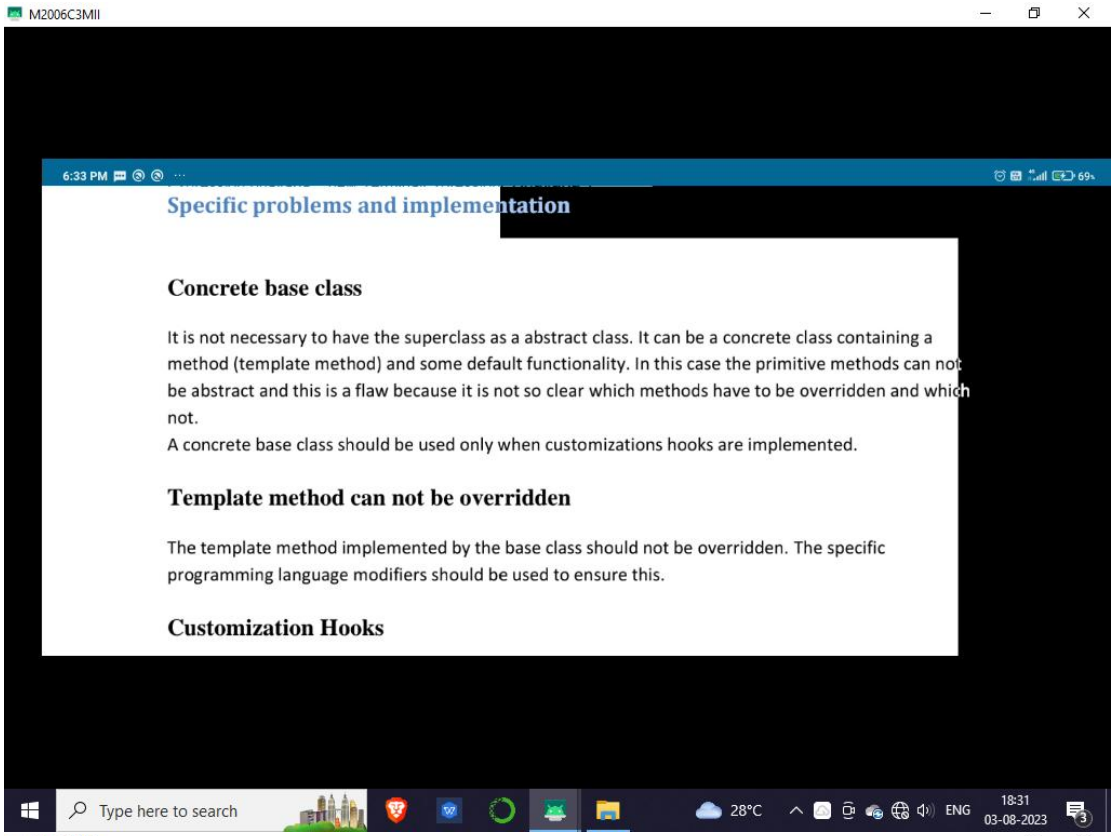
```
public class Trip {
    public final void performTrip(){
        doComingTransport();
        doDayA();
        doDayB();
        doDayC();
        doReturningTransport
    }
```

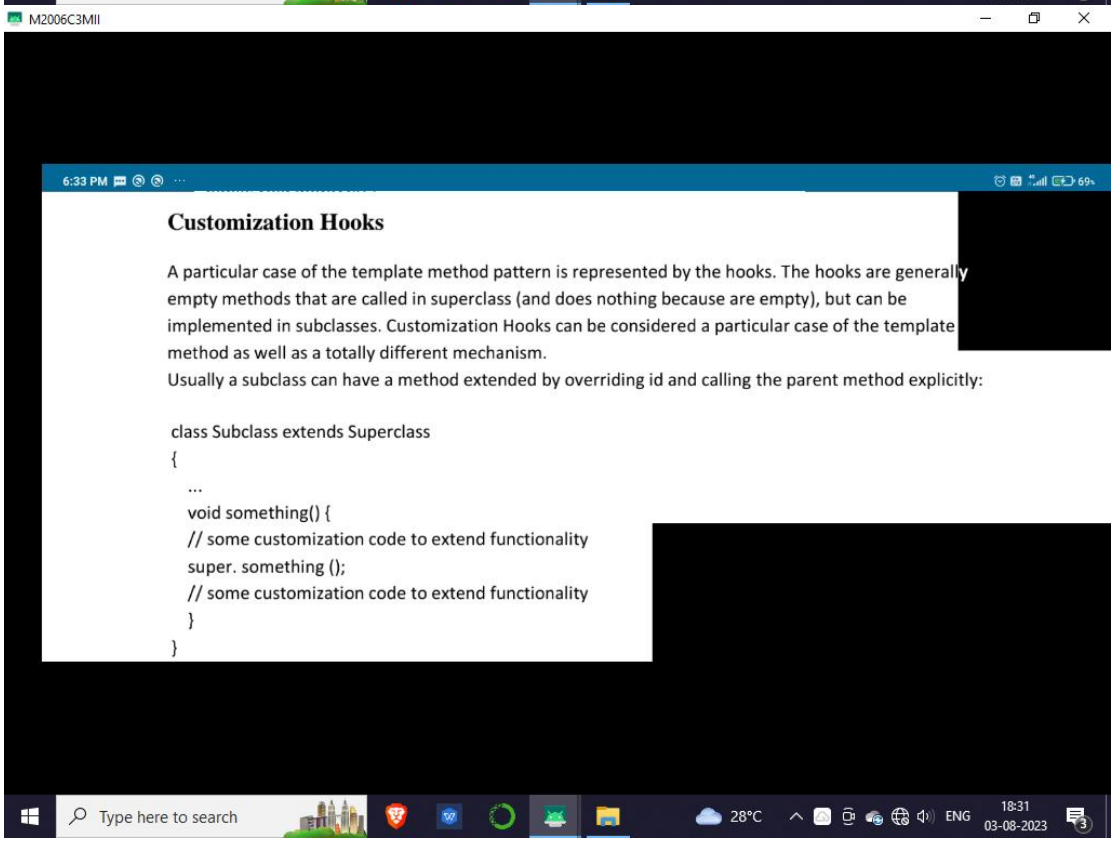

M2006C3MII
6:33 PM
trip. All the trips contain common behavior but there are several packages. For example each trip
contains the basic steps:
- The tourists are transported to the holiday location by plane/train/ships,...
- Each day they are visiting something
- They are returning back home.
So we create an abstract class containing each step as an abstract method and one concrete and final
method that calls all the abstracts methods. Then we create one superclass for each package:
public class Trip {
public final void performTrip(){
doComingTransport();
doDayA();
doDayB();
doDayC();
doReturningTransport
}
public abstract void doComingTransport();
Type here to search
28°C
ENG
18:31
03-08-2023


M2006C3MII
6:33 PM
public abstract void doDayA();
public abstract void doDayB();
public abstract void doDayC();
public abstract void doReturningTransport();
}
public class PackageA extends Trip {
public void doComingTransport() {
System.out.println("The turists are comming by air ...");
}
public void doDayA() {
System.out.println("The turists are visiting the aquarium ...");
}
public void doDayB() {
System.out.println("The turists are going to the beach ...");
}
public void doDayC() {
Type here to search
28°C
ENG
18:31
03-08-2023


M2006C3MII
6:33 PM
}
public void doDayB() {
System.out.println("The turists are going to the beach ...");
}
public void doDayC() {
System.out.println("The turists are going to mountains ...");
}
public void doReturningTransport() {
System.out.println("The turists are going home by air ...");
}
}
public class PackageB extends Trip {
public void doComingTransport() {
System.out.println("The turists are comming by train ...");
}
public void doDayA() {
System.out.println("The turists are visiting the mountain ...");
}
Type here to search
28°C
ENG
18:31
03-08-2023


M2006C3MII
6:33 PM
}
public void doDayA() {
System.out.println("The turists are visiting the mountain ...");
}
public void doDayB() {
System.out.println("The turists are going to the beach ...");
}
public void doDayC() {
System.out.println("The turists are going to zoo ...");
}
public void doReturningTransport() {
System.out.println("The turists are going home by train ...");
}
}
Type here to search
28°C
ENG
18:31
03-08-2023


M2006C3MII
6:33 PM
Specific problems and implementation
Concrete base class
It is not necessary to have the superclass as a abstract class. It can be a concrete class containing a method (template method) and some default functionality. In this case the primitive methods can not be abstract and this is a flaw because it is not so clear which methods have to be overridden and which not.
A concrete base class should be used only when customizations hooks are implemented.
Template method can not be overridden
The template method implemented by the base class should not be overridden. The specific programming language modifiers should be used to ensure this.
Customization Hooks
Type here to search
28°C
ENG
18:31
03-08-2023


M2006C3MII
6:33 PM
Customization Hooks
A particular case of the template method pattern is represented by the hooks. The hooks are generally empty methods that are called in superclass (and does nothing because are empty), but can be implemented in subclasses. Customization Hooks can be considered a particular case of the template method as well as a totally different mechanism.
Usually a subclass can have a method extended by overriding id and calling the parent method explicitly:
class Subclass extends Superclass
{
...
void something() {
// some customization code to extend functionality
super. something ();
// some customization code to extend functionality
}
}
Type here to search
28°C
ENG
18:31
03-08-2023

M2006C3MII

6:33 PM

```
...
void something() {
// some customization code to extend functionality
super. something ();
// some customization code to extend functionality
}
}
```

Unfortunately it is easy to forget to call the super and this is forcing the developer to check the existing code from the method in Superclass.
Instead of overriding some hook methods can be added. Then in the subclasses only the hooks should be implemented without being aware of the method something:

```
class Superclass
{
```

Type here to search 28°C ENG 18:32 03-08-2023

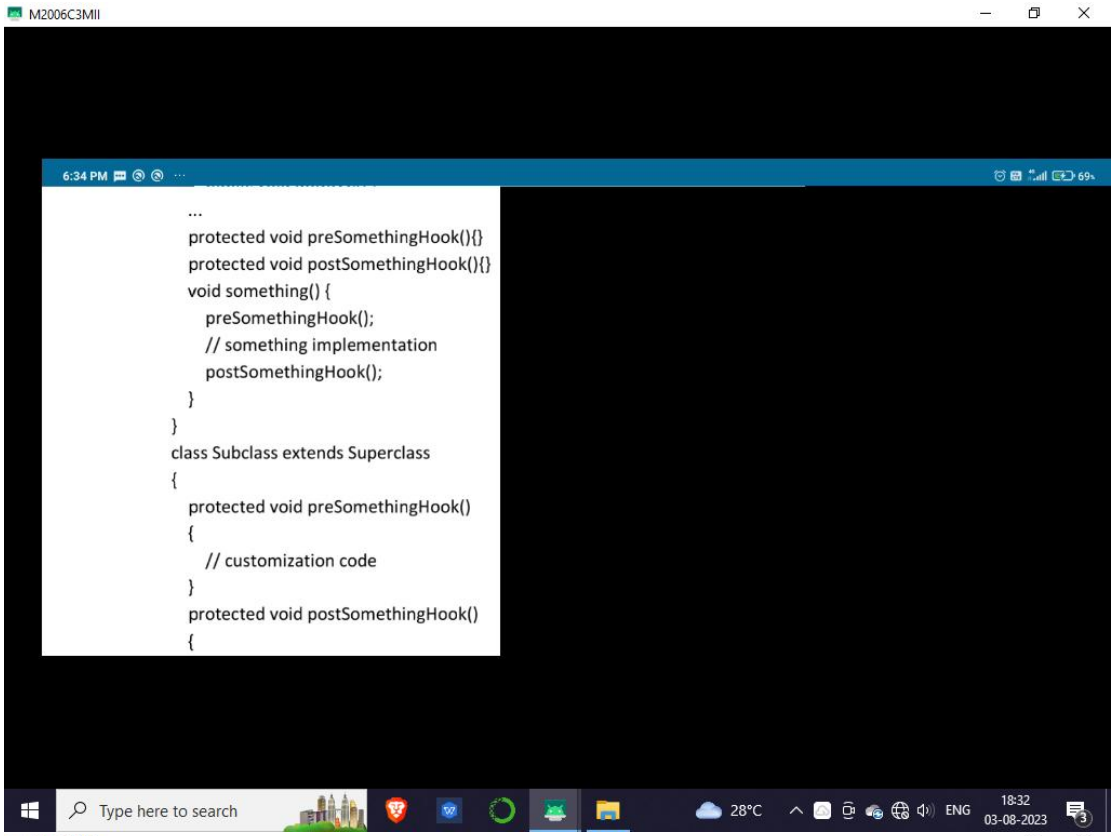
M2006C3MII

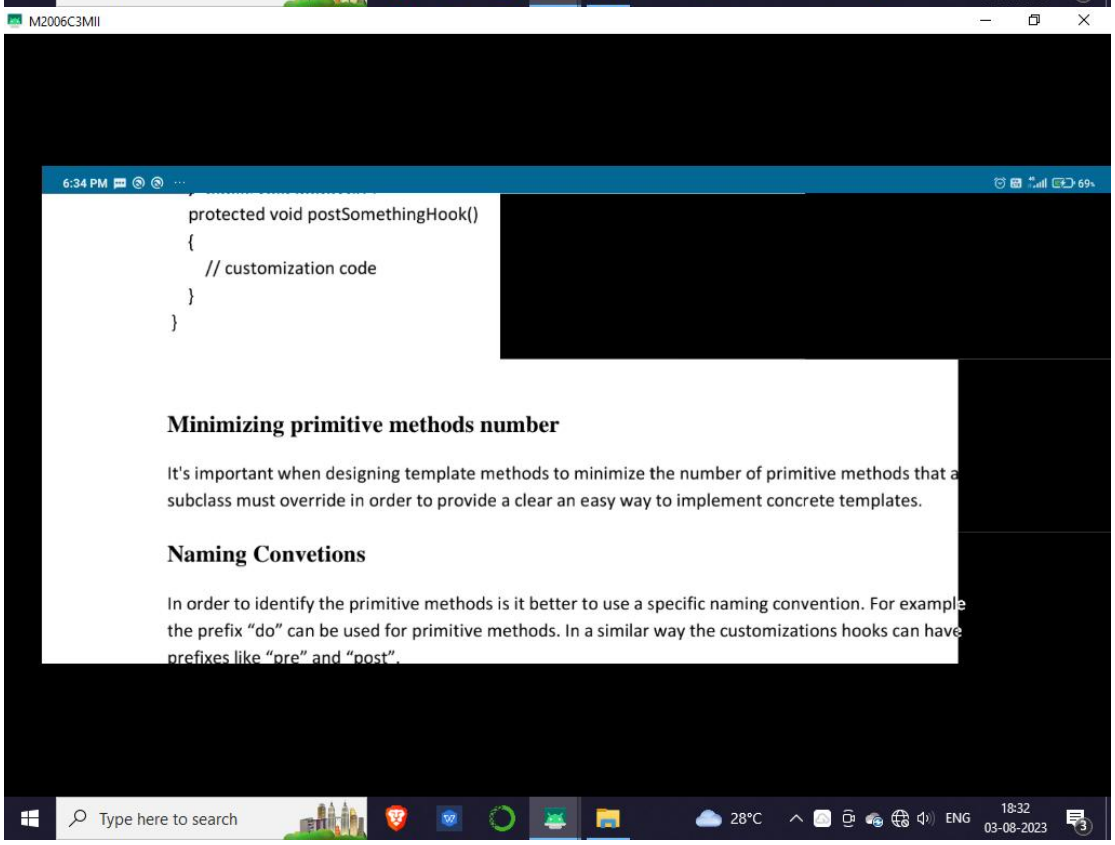
6:34 PM

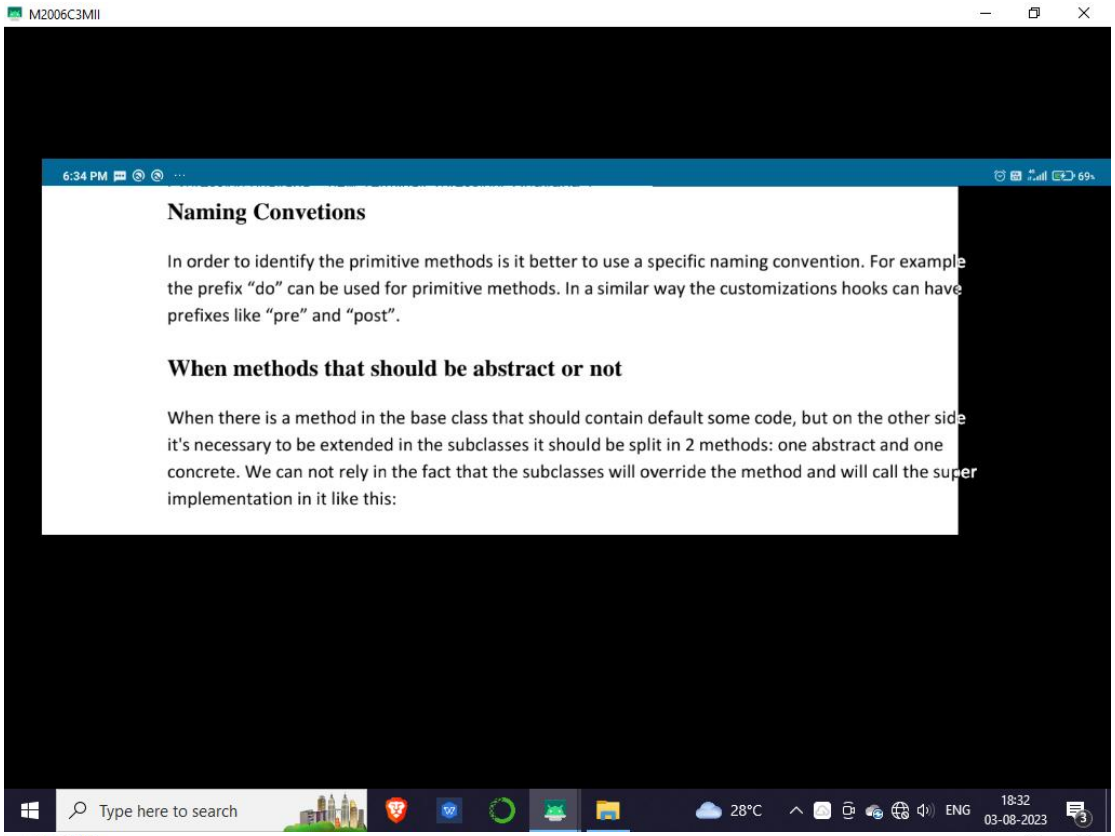
```
class Superclass
{
```

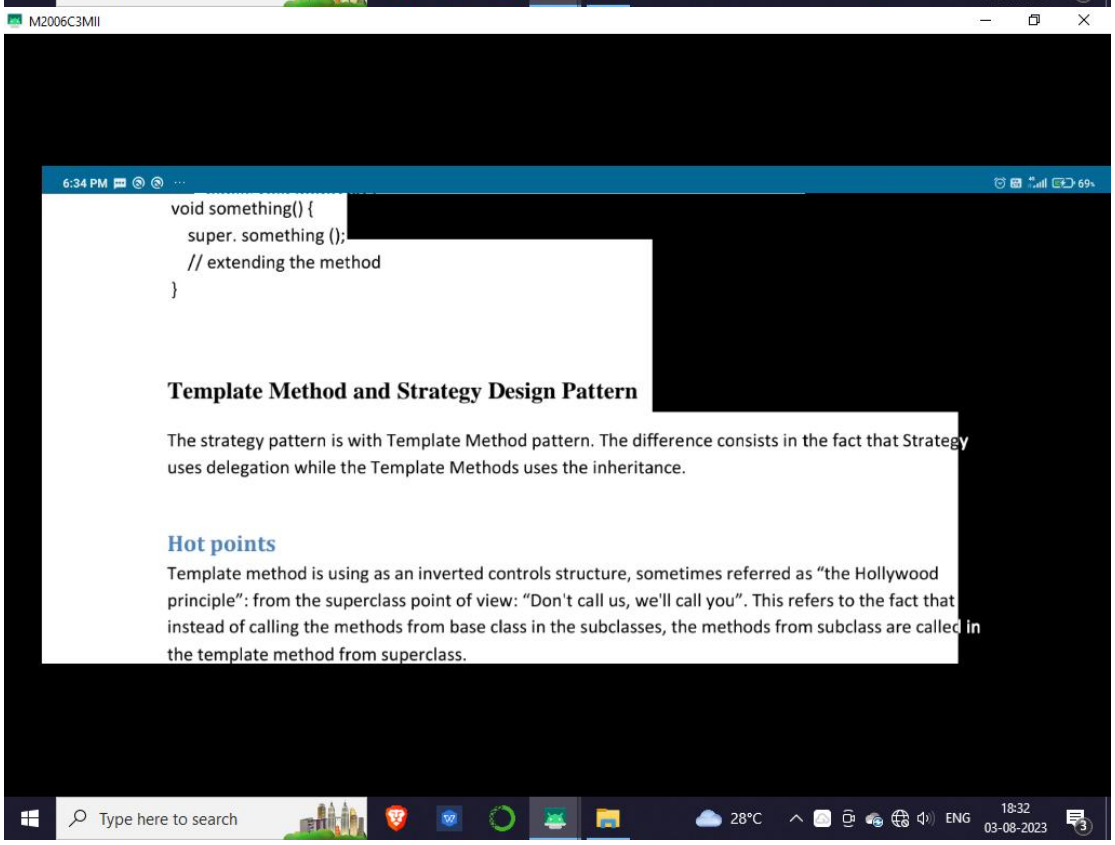
```
...
protected void preSomethingHook(){}
protected void postSomethingHook(){}
void something() {
  preSomethingHook();
```

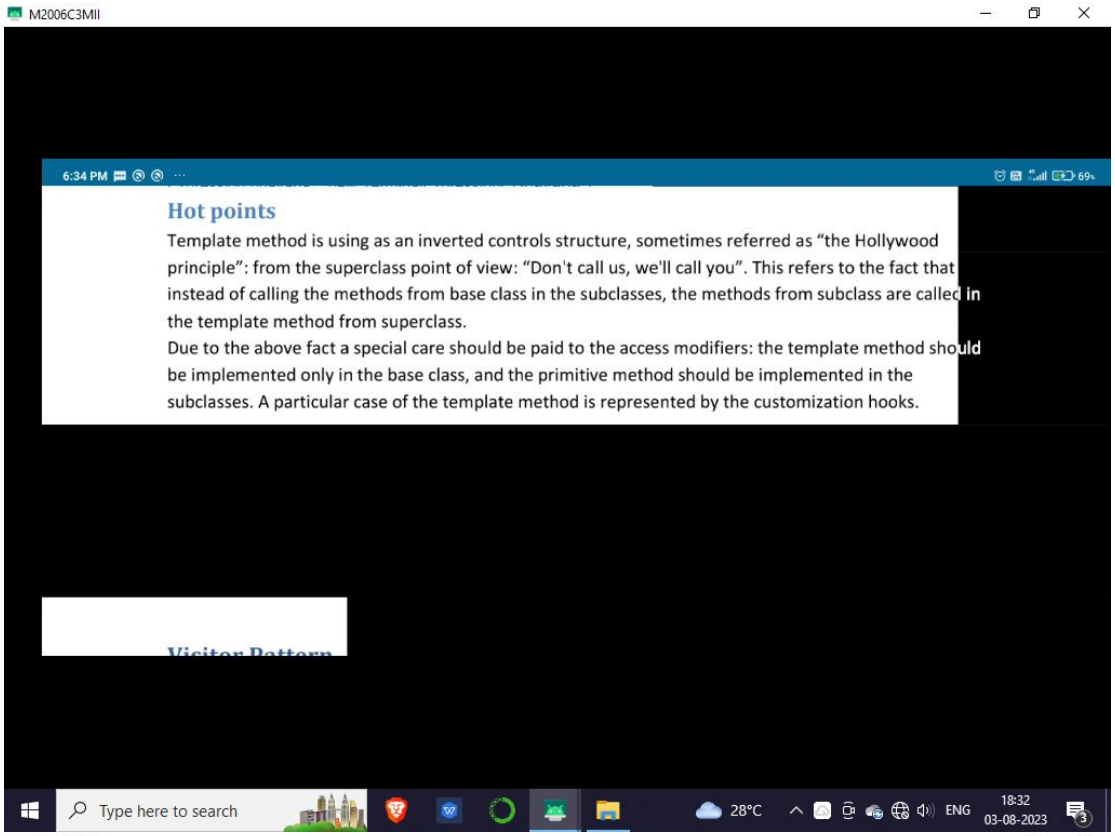
Type here to search 28°C ENG 18:32 03-08-2023

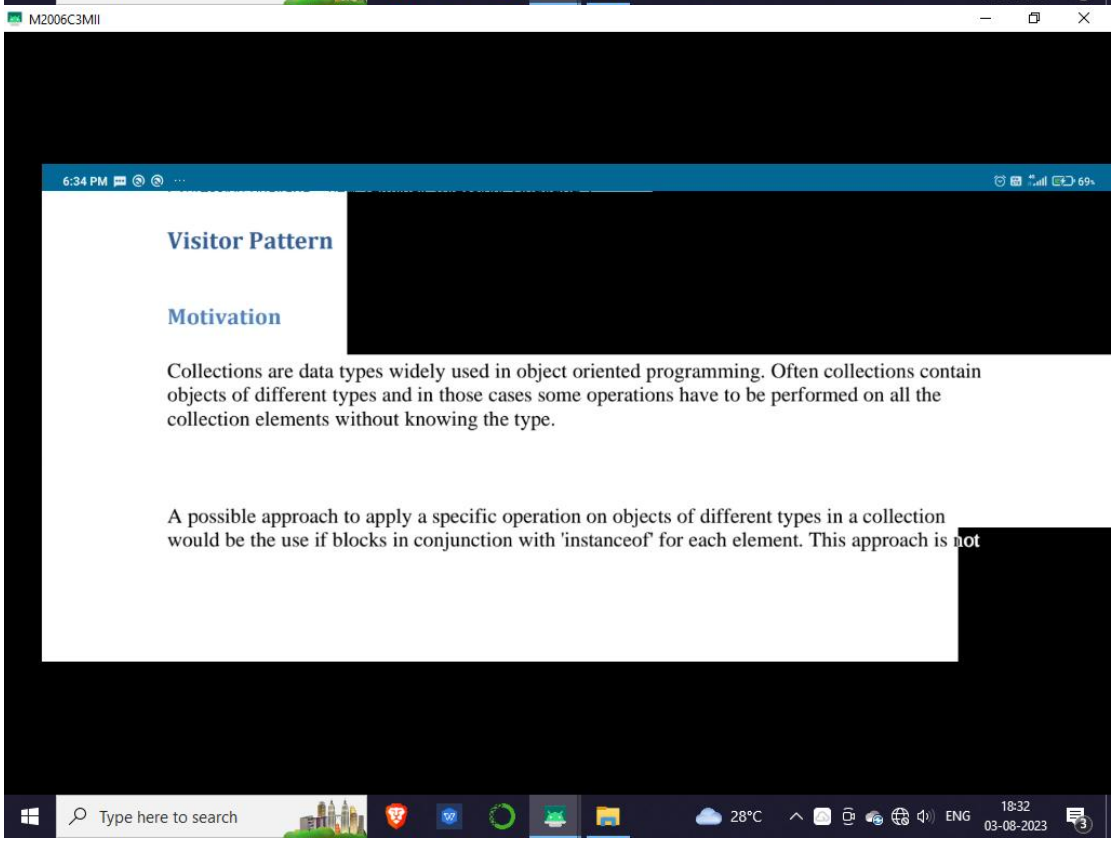

M2006C3MII
6:34 PM
...
protected void preSomethingHook(){}
protected void postSomethingHook(){}
void something() {
preSomethingHook();
// something implementation
postSomethingHook();
}
}
class Subclass extends Superclass
{
protected void preSomethingHook()
{
// customization code
}
protected void postSomethingHook()
{
Type here to search
28°C
ENG
18:32
03-08-2023


M2006C3MII
6:34 PM
protected void postSomethingHook()
{
// customization code
}
}
Minimizing primitive methods number
It's important when designing template methods to minimize the number of primitive methods that a
subclass must override in order to provide a clear an easy way to implement concrete templates.
Naming Convetions
In order to identify the primitive methods is it better to use a specific naming convention. For example
the prefix "do" can be used for primitive methods. In a similar way the customizations hooks can have
prefixes like "pre" and "post".
Type here to search
28°C
ENG
18:32
03-08-2023


M2006C3MII
6:34 PM
Naming Convetions
In order to identify the primitive methods is it better to use a specific naming convention. For example the prefix “do” can be used for primitive methods. In a similar way the customizations hooks can have prefixes like “pre” and “post”.
When methods that should be abstract or not
When there is a method in the base class that should contain default some code, but on the other side it's necessary to be extended in the subclasses it should be split in 2 methods: one abstract and one concrete. We can not rely in the fact that the subclasses will override the method and will call the super implementation in it like this:
Type here to search
28°C
ENG
18:32
03-08-2023


M2006C3MII
6:34 PM
void something() {
super. something ();
// extending the method
}
Template Method and Strategy Design Pattern
The strategy pattern is with Template Method pattern. The difference consists in the fact that Strategy uses delegation while the Template Methods uses the inheritance.
Hot points
Template method is using as an inverted controls structure, sometimes referred as “the Hollywood principle”: from the superclass point of view: “Don't call us, we'll call you”. This refers to the fact that instead of calling the methods from base class in the subclasses, the methods from subclass are called in the template method from superclass.
Type here to search
28°C
ENG
18:32
03-08-2023


M2006C3MII
6:34 PM
Hot points
Template method is using as an inverted controls structure, sometimes referred as “the Hollywood principle”: from the superclass point of view: “Don't call us, we'll call you”. This refers to the fact that instead of calling the methods from base class in the subclasses, the methods from subclass are called in the template method from superclass.
Due to the above fact a special care should be paid to the access modifiers: the template method should be implemented only in the base class, and the primitive method should be implemented in the subclasses. A particular case of the template method is represented by the customization hooks.
Visitor Pattern
Type here to search
28°C
ENG
18:32
03-08-2023


M2006C3MII
6:34 PM
Visitor Pattern
Motivation
Collections are data types widely used in object oriented programming. Often collections contain objects of different types and in those cases some operations have to be performed on all the collection elements without knowing the type.
A possible approach to apply a specific operation on objects of different types in a collection would be the use if blocks in conjunction with 'instanceof' for each element. This approach is not
Type here to search
28°C
ENG
18:32
03-08-2023

M2006C3MII

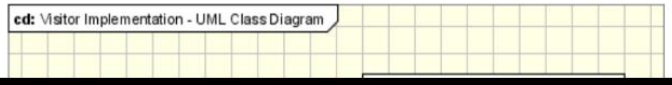
6:34 PM

a nice one, not flexible and not object oriented at all. At this point we should think to the Open Close principle and we should remember from there that we can replace if blocks with an abstract class and each concrete class will implement its own operation.

Intent

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

Implementation

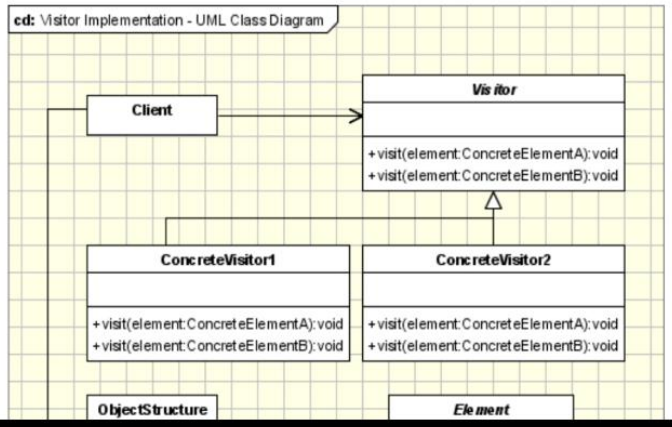


Type here to search 28°C ENG 18:32 03-08-2023

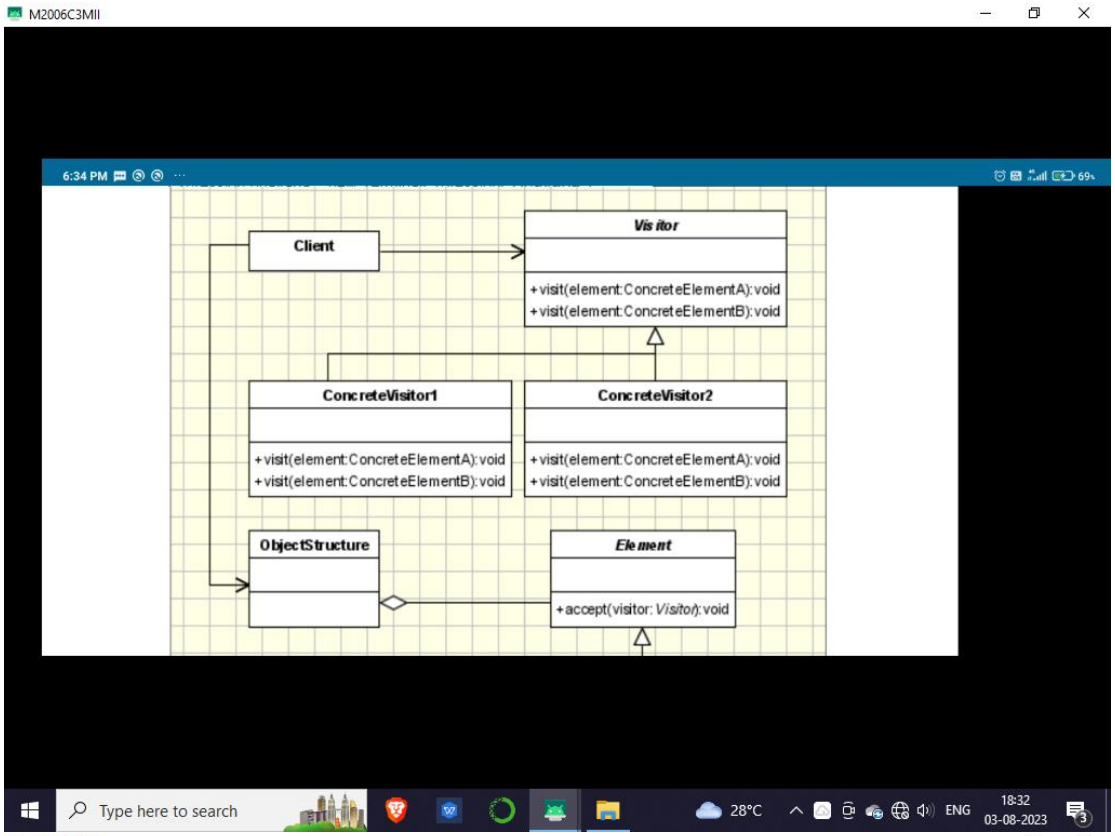
M2006C3MII

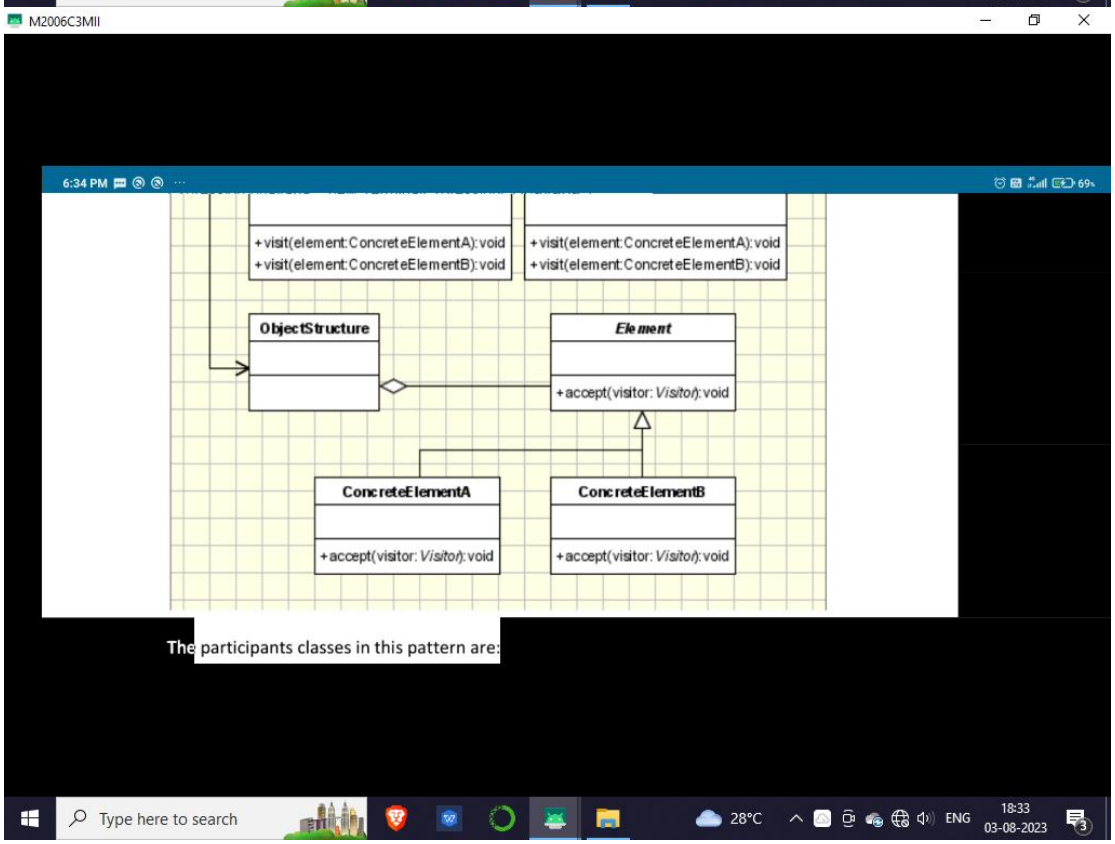
6:34 PM

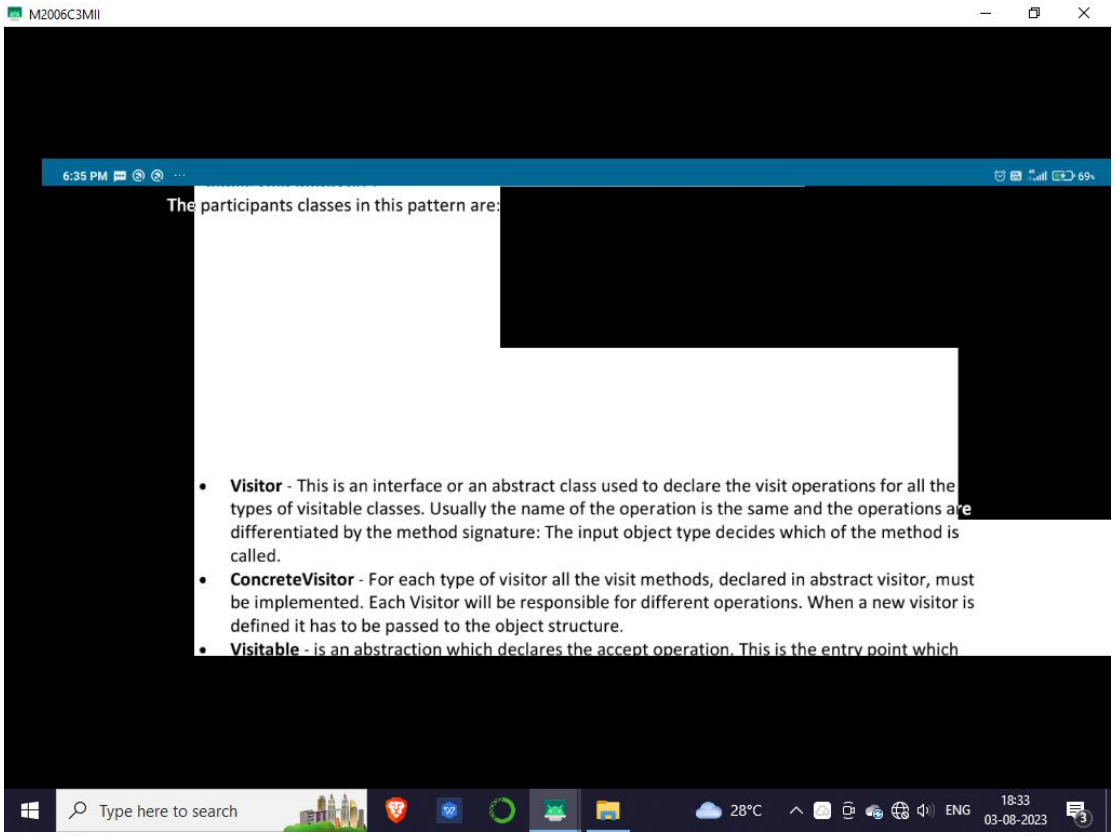
Implementation

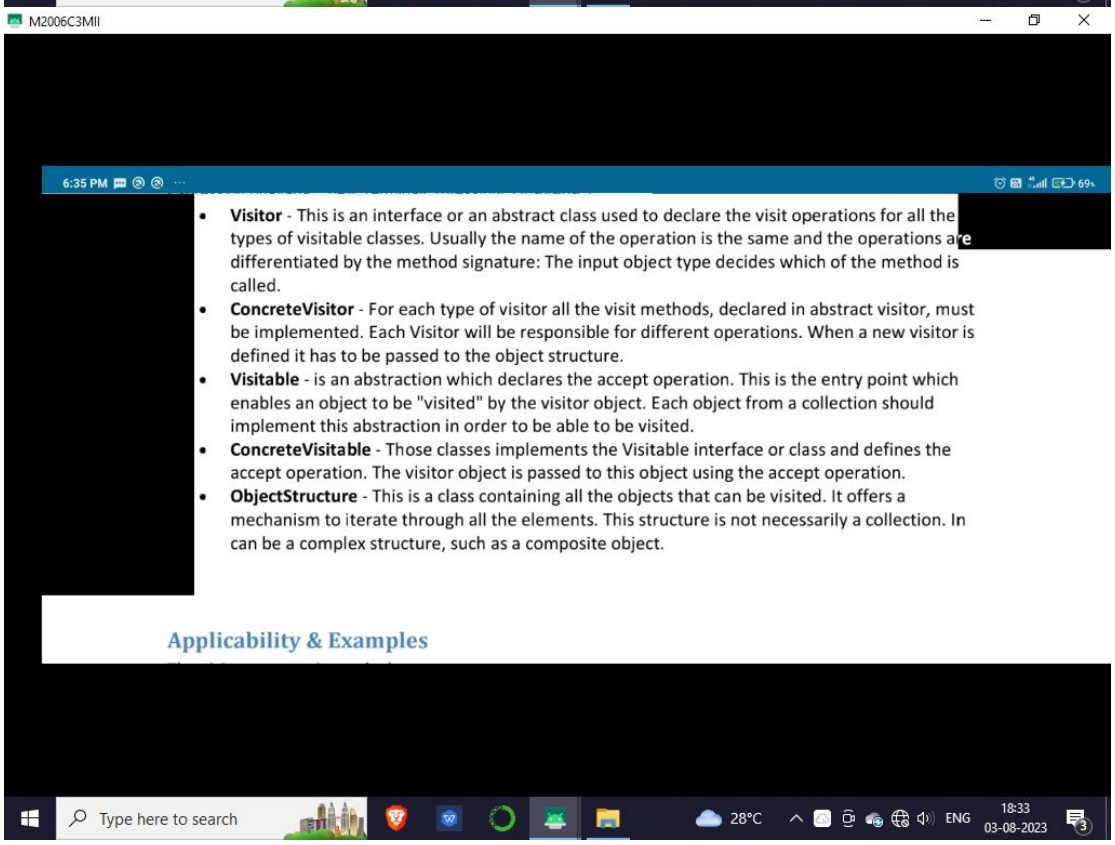


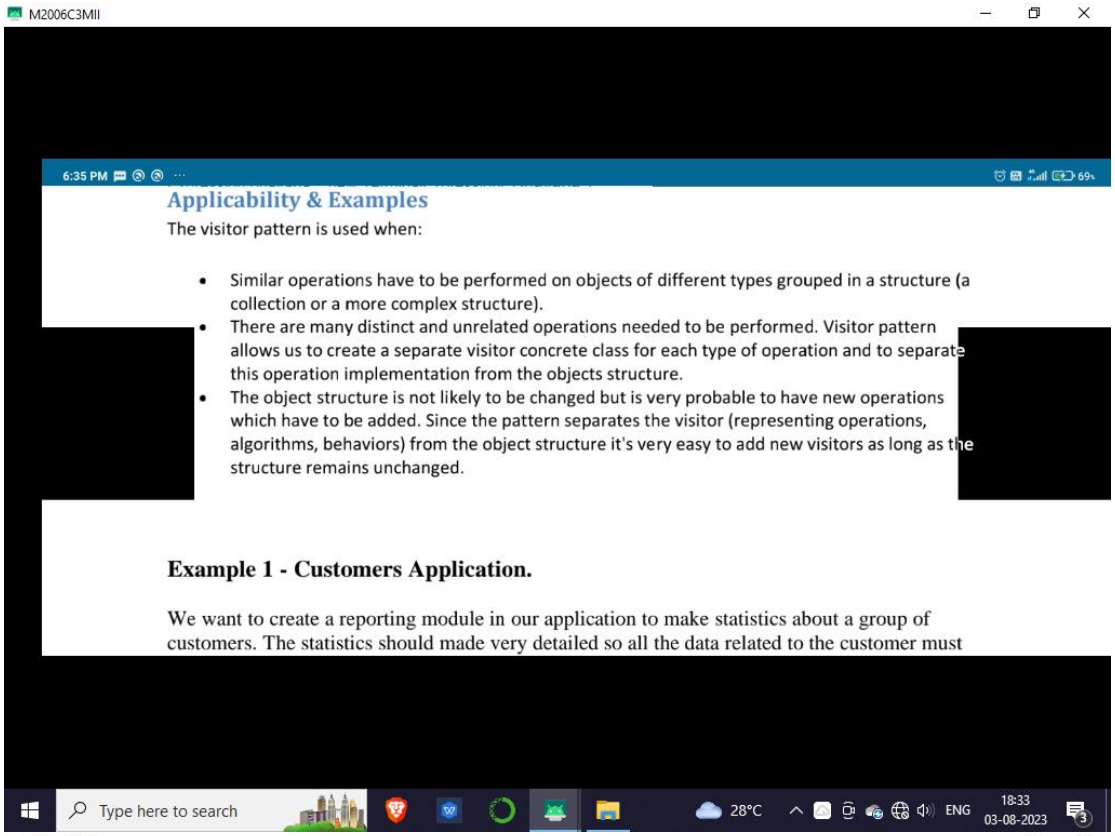
Type here to search 28°C ENG 18:32 03-08-2023

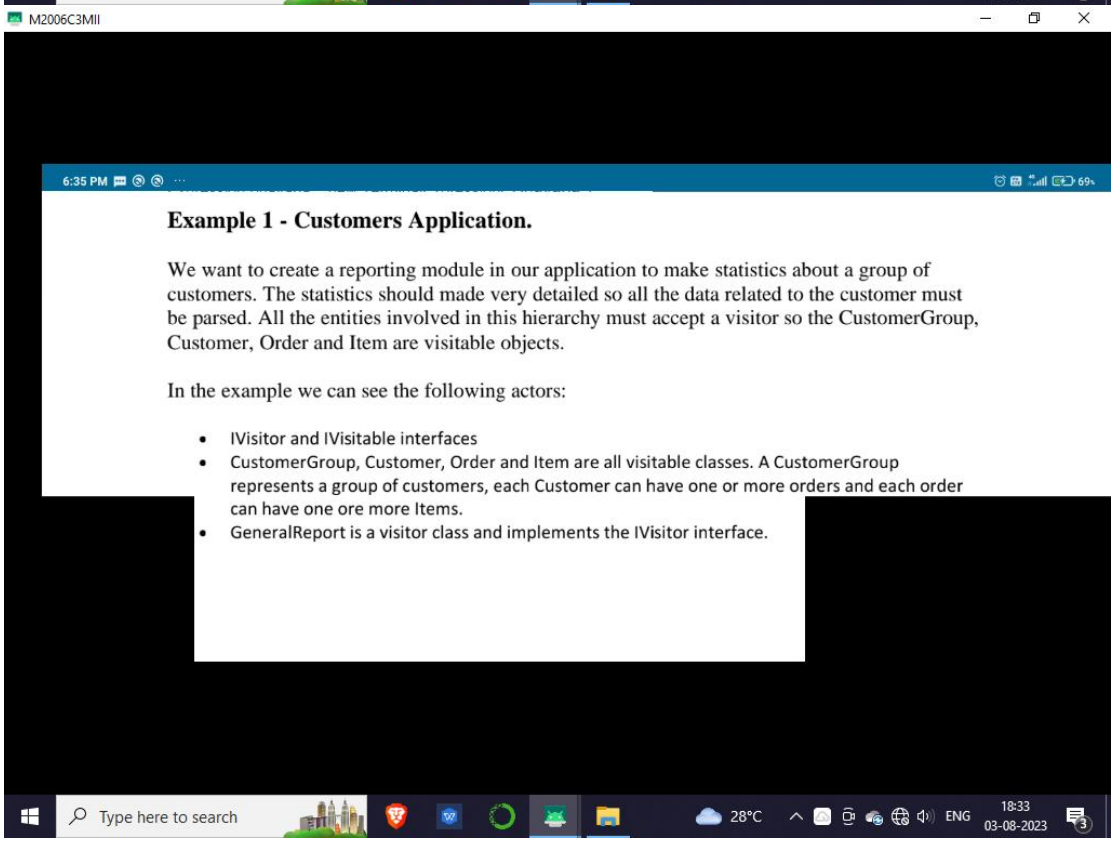

M2006C3MII
6:34 PM
Client
Visitor
+visit(element:ConcreteElementA):void
+visit(element:ConcreteElementB):void
ConcreteVisitor1
+visit(element:ConcreteElementA):void
+visit(element:ConcreteElementB):void
ConcreteVisitor2
+visit(element:ConcreteElementA):void
+visit(element:ConcreteElementB):void
ObjectStructure
Element
+accept(visitor: Visitor):void
Type here to search
28°C
ENG
18:32
03-08-2023


M2006C3MII
6:34 PM
+visit(element:ConcreteElementA):void
+visit(element:ConcreteElementB):void
+visit(element:ConcreteElementA):void
+visit(element:ConcreteElementB):void
ObjectStructure
Element
+accept(visitor: Visitor):void
ConcreteElementA
+accept(visitor: Visitor):void
ConcreteElementB
+accept(visitor: Visitor):void
The participants classes in this pattern are:
Type here to search
28°C
ENG
18:33
03-08-2023


M2006C3MII
6:35 PM
The participants classes in this pattern are:
Visitor - This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes. Usually the name of the operation is the same and the operations are differentiated by the method signature: The input object type decides which of the method is called.
ConcreteVisitor - For each type of visitor all the visit methods, declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations. When a new visitor is defined it has to be passed to the object structure.
Visitable - is an abstraction which declares the accept operation. This is the entry point which
Type here to search
28°C
ENG
18:33
03-08-2023


M2006C3MII
6:35 PM
Visitor - This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes. Usually the name of the operation is the same and the operations are differentiated by the method signature: The input object type decides which of the method is called.
ConcreteVisitor - For each type of visitor all the visit methods, declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations. When a new visitor is defined it has to be passed to the object structure.
Visitable - is an abstraction which declares the accept operation. This is the entry point which enables an object to be "visited" by the visitor object. Each object from a collection should implement this abstraction in order to be able to be visited.
ConcreteVisitable - Those classes implements the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.
ObjectStructure - This is a class containing all the objects that can be visited. It offers a mechanism to iterate through all the elements. This structure is not necessarily a collection. In can be a complex structure, such as a composite object.
Applicability & Examples
Type here to search
28°C
ENG
18:33
03-08-2023

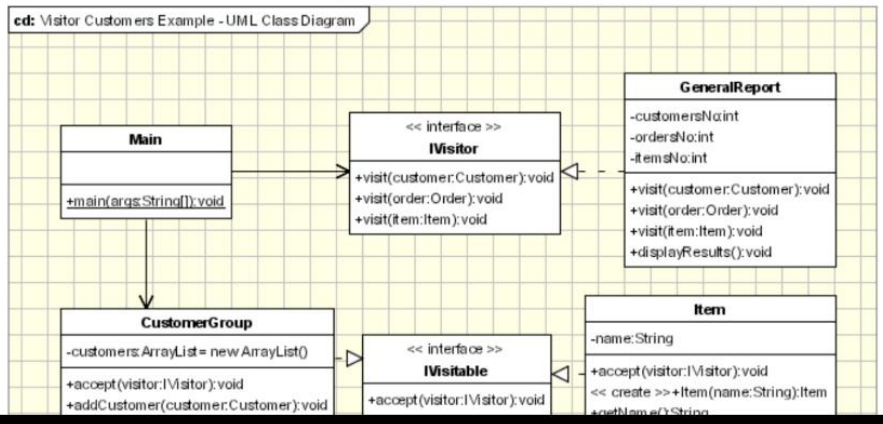

M2006C3MII
6:35 PM
Applicability & Examples
The visitor pattern is used when:
Similar operations have to be performed on objects of different types grouped in a structure (a collection or a more complex structure).
There are many distinct and unrelated operations needed to be performed. Visitor pattern allows us to create a separate visitor concrete class for each type of operation and to separate this operation implementation from the objects structure.
The object structure is not likely to be changed but is very probable to have new operations which have to be added. Since the pattern separates the visitor (representing operations, algorithms, behaviors) from the object structure it's very easy to add new visitors as long as the structure remains unchanged.
Example 1 - Customers Application.
We want to create a reporting module in our application to make statistics about a group of customers. The statistics should made very detailed so all the data related to the customer must
Type here to search
28°C
ENG
18:33
03-08-2023


M2006C3MII
6:35 PM
Example 1 - Customers Application.
We want to create a reporting module in our application to make statistics about a group of customers. The statistics should made very detailed so all the data related to the customer must be parsed. All the entities involved in this hierarchy must accept a visitor so the CustomerGroup, Customer, Order and Item are visitable objects.
In the example we can see the following actors:
IVisitor and IVisitable interfaces
CustomerGroup, Customer, Order and Item are all visitable classes. A CustomerGroup represents a group of customers, each Customer can have one or more orders and each order can have one ore more Items.
GeneralReport is a visitor class and implements the IVisitor interface.
Type here to search
28°C
ENG
18:33
03-08-2023

M2006C3MII

6:35 PM

Source: Click here to see java source code

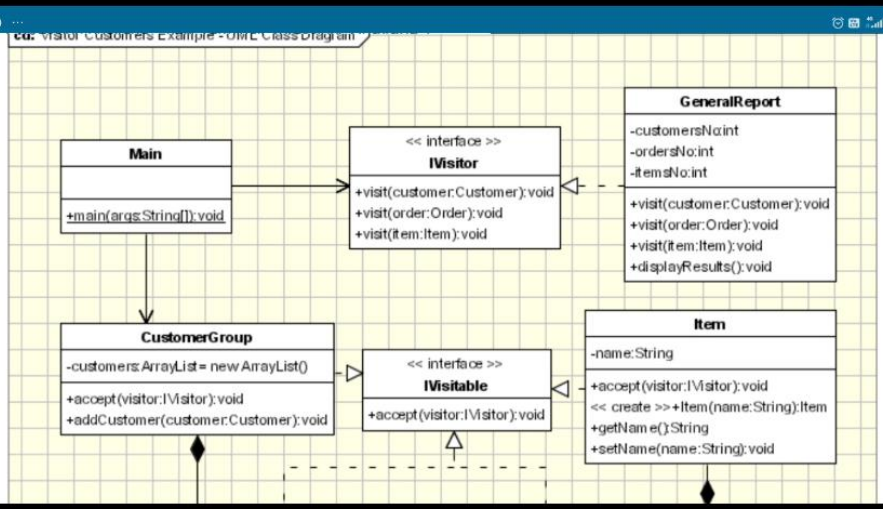


Type here to search

28°C ENG 18:33 03-08-2023

M2006C3MII

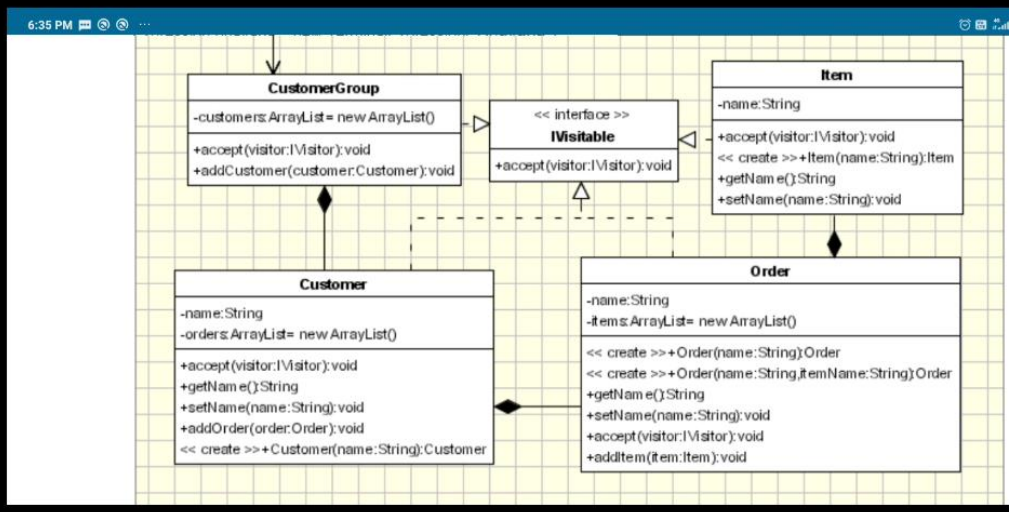
6:35 PM



Type here to search

28°C ENG 18:33 03-08-2023

M2006C3MII



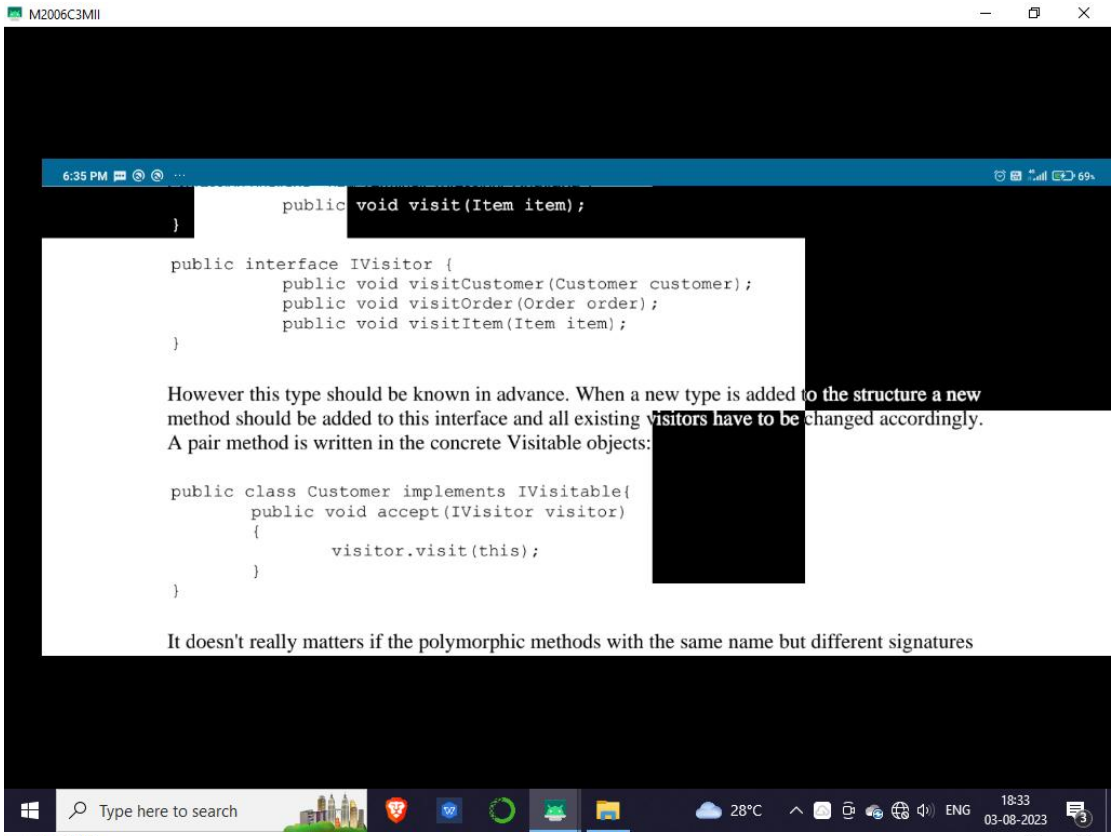
M2006C3MII

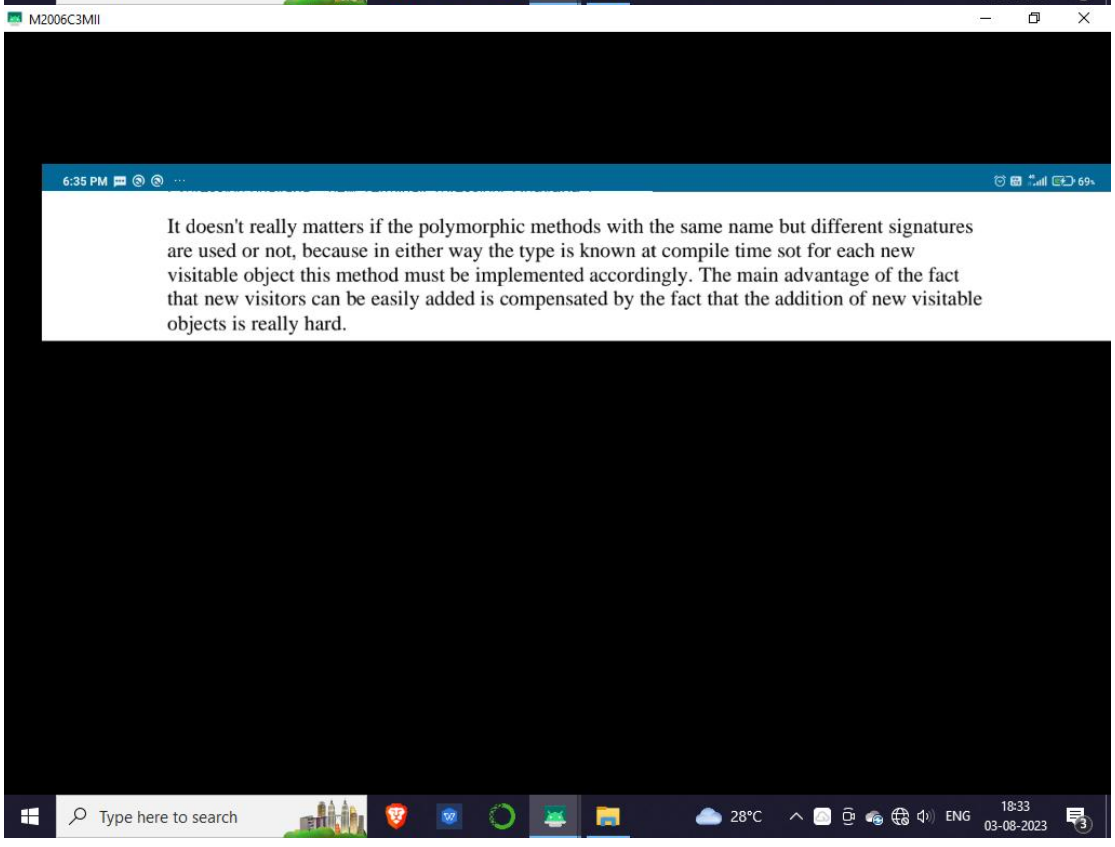
Specific problems and implementation

Tight Coupled Visitable objects

The classic implementation of the Visitor pattern have a major drawback because the type of visitor methods has to be known in advance. The Visitor interface can be defined using polymorphic methods or methods with different names:

```
public interface IVisitor {
        public void visit(Customer customer);
        public void visit(Order order);
```


M2006C3MII
6:35 PM
public void visit(Item item);
}
public interface IVisitor {
public void visitCustomer(Customer customer);
public void visitOrder(Order order);
public void visitItem(Item item);
}
However this type should be known in advance. When a new type is added to the structure a new method should be added to this interface and all existing visitors have to be changed accordingly. A pair method is written in the concrete Visitable objects:
public class Customer implements IVisitable{
public void accept(IVisitor visitor)
{
visitor.visit(this);
}
}
It doesn't really matters if the polymorphic methods with the same name but different signatures
Type here to search
28°C
ENG
18:33
03-08-2023


M2006C3MII
6:35 PM
It doesn't really matters if the polymorphic methods with the same name but different signatures are used or not, because in either way the type is known at compile time sot for each new visitable object this method must be implemented accordingly. The main advantage of the fact that new visitors can be easily added is compensated by the fact that the addition of new visitable objects is really hard.
Type here to search
28°C
ENG
18:33
03-08-2023