VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

M.Tech Software Engineering - Win Sem -2018-19

SWE2019-Design Patterns

FINAL REPORT

ATM Money Handler Using Chain of Responsibility Design Pattern

Submitted by:

16MIS0433 - Mano Bala.R

16MIS0210 - Dhivakaran.S

Faculty : Senthil Kumaran U

Slot : A1

1.Introduction :

Let's suppose an user wants to withdraw some amount of money from the nearest ATM. Here in the above figure an user (John) inserts his debit card into the ATM machine and enter his pin number and amount of money he wants. John enters 455 rupees in place of amount field then ATM sends request to hundred rupees handler it will give four pieces of hundred rupees notes. For remaining 55 rupees hundred rupees handler sends request to fifty rupees handler it gives one piece of fifty rupees note then fifty rupees handler sends request to five rupees handler it gives one five rupees note. The twenty rupees handler will not do any operation as there is no requirement of twenty rupees.

2.Solution:

This pattern is useful for modelling requests and handling events if we do not know the number and type of handlers in advance. The event – based systems, purchasing systems and shipping systems, ATM, Bill Payment Kiosk are real life examples that places well with the chain of responsibility pattern. Thus it concludes that Chain of Responsibility pattern allows multiple objects in a chain to handle a request object. The request flows through the chain until it is handled by a link in the chain. The chain of responsibilities design pattern is a very powerful pattern. A chain of chains can also be implemented, creating a multi- dimensional structure.

3.Chain of Responsibility :

As the name suggests, the chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request. This pattern comes under behavioral patterns.

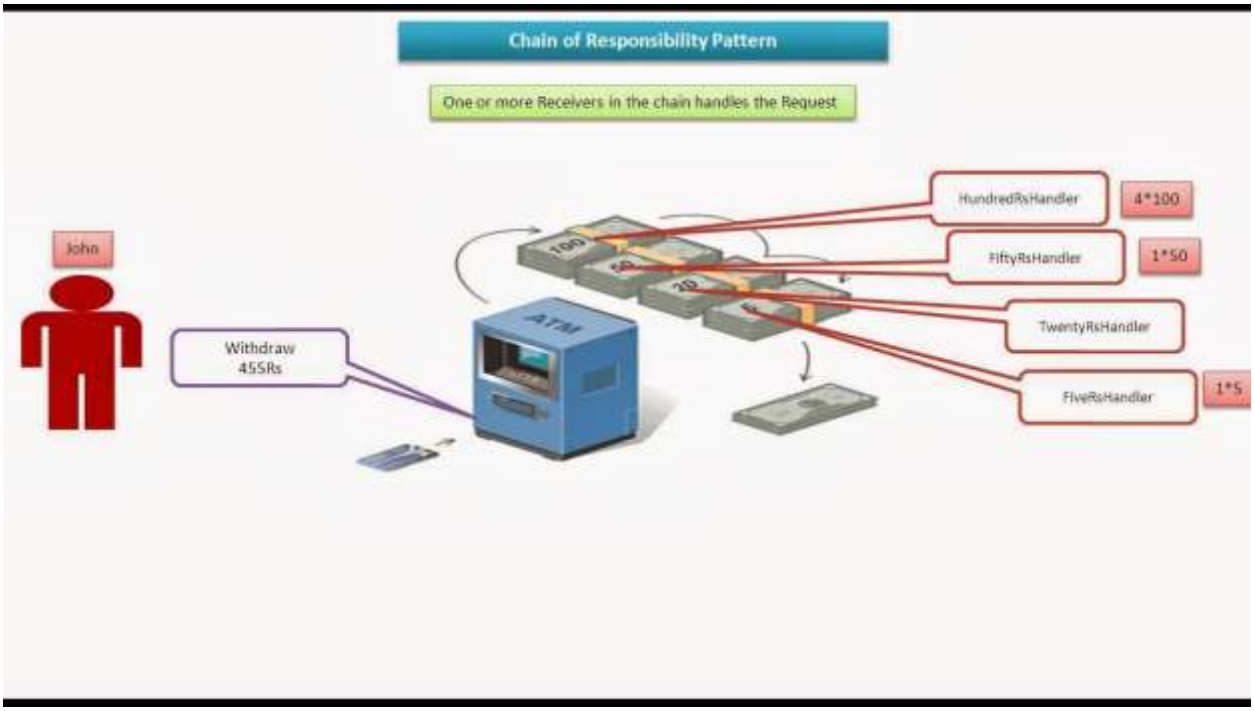
In this pattern, normally each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

4.Intent :

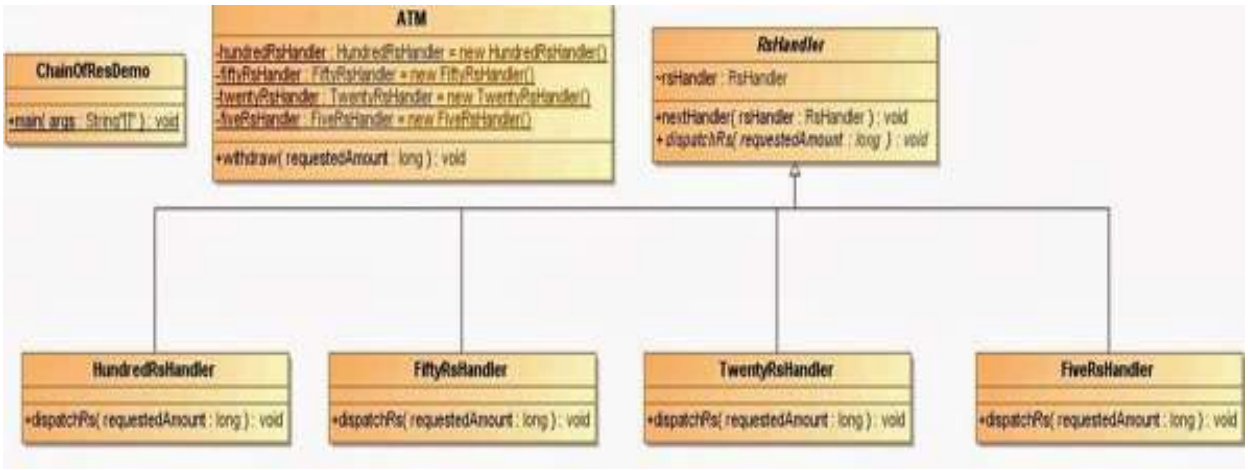
- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- Chain the receiving objects and pass the request along the chain until an object handles it.
- It avoids attaching the sender of a request to its receiver, giving this way other objects 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.

5.Motivation:

In this scenario we take an example of ATM machine. Let's suppose an user wants to withdraw some amount of money from the nearest ATM. Here in the above figure an user (John) inserts his debit card into the ATM machine and enter his pin number and amount of money he wants. John enters 455 rupees in place of amount field then ATM sends request to hundred rupees handler it will give four pieces of hundred rupees notes. For remaining 55 rupees hundred rupees handler sends request to fifty rupees handler it gives one piece of fifty rupees note then fifty rupees handler sends request to five rupees handler it gives one five rupees note. The twenty rupees handler will not do any operation as there is no requirement of twenty rupees. Here in the above diagram rupees handlers are working as a chain of objects.



- Consider the Button, Dialog, and Application classes use HelpHandler operations to handle help requests.
- HelpHandler's Handle Help operation forwards the request to the successor by default.
- Subclasses can override this operation to provide help under the right circumstances; otherwise they can use the default implementation to forward the request.

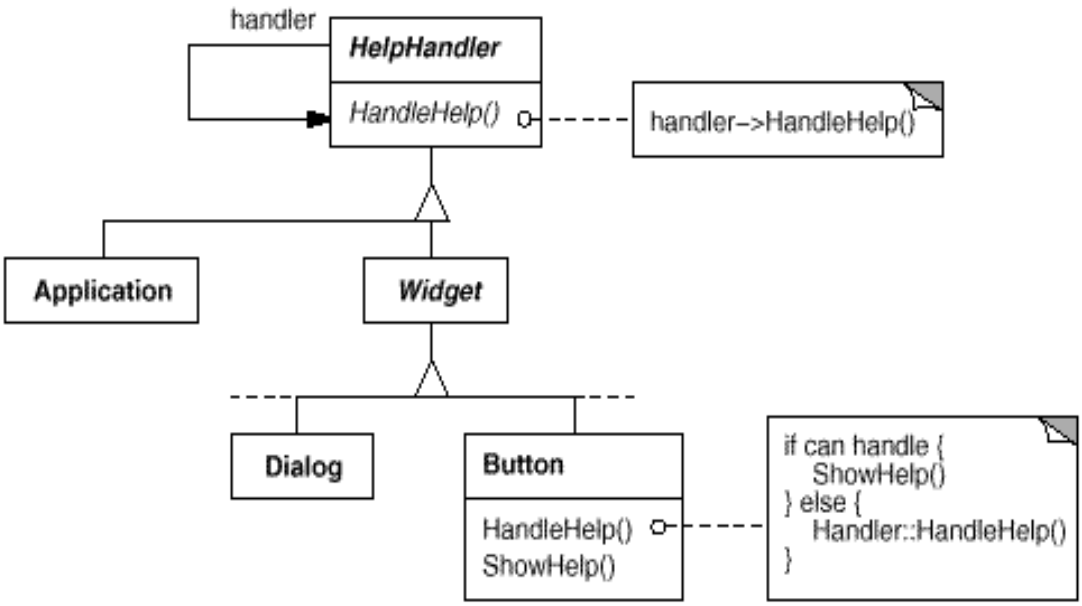


6.Applicability :

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 may handle a request, and the handler isn't known *apriori. The handler should be ascertained automatically.*
- you want to issue a request to one of several objects without specifying the receiver explicitly.
- the set of objects that can handle a request should be specified dynamically.
- 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

7.Structure Diagram :

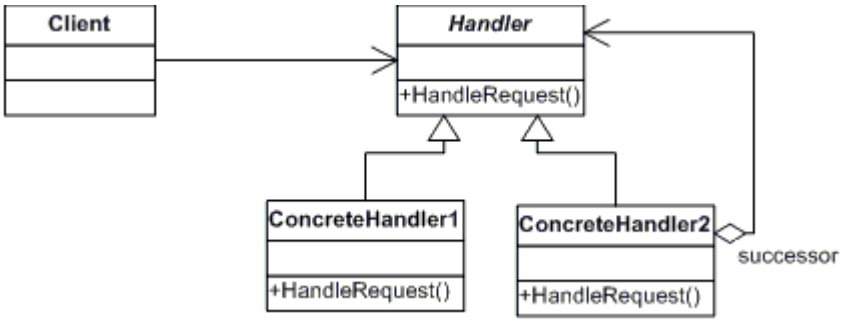


8.Participants :

1) Handler : This can be an interface which will primarily recieve the request and dispatches the request to chain of handlers. It has reference of only first handler in the chain and does not know anything about rest of the handlers.

2) Concrete handlers : These are actual handlers of the request chained in some sequential order.

3) Client : Originator of request and this will access the handler to handle it.



Participants in chain of responsibility

9.Consequences :

Chain of Responsibility has the following benefits and liabilities:

- 1. *Reduced coupling.*
- 2. *Added flexibility in assigning responsibilities to objects.*
- 3. *Receipt isn't guaranteed.*

10.Collaborations :

- When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

11.Implementation :

Code :

RsHandler.java

```
public abstract class RsHandler
{
 RsHandler rsHandler;
 public void nextHandler( RsHandler rsHandler )
 {
 this.rsHandler = rsHandler;
 }
 public abstract void dispatchRs( long requestedAmount );
}
```

HundredRsHandler.java

```
public class HundredRsHandler extends RsHandler
{
```

```
 public void dispatchRs(long requestedAmount)
 {
 long numberofNotesToBeDispatched = requestedAmount / 100;
 if (numberofNotesToBeDispatched > 0)
 {
 if(numberofNotesToBeDispatched>1)
 {
 System.out.println(numberofNotesToBeDispatched + " ,Hundred Rs notes are
dispatched by HundredRsHandler\n");
 }
 else
 {
 System.out.println(numberofNotesToBeDispatched + " ,Hundred Rs note is dispatched
by HundredRsHandler\n");
 }
 }
 long pendingAmountToBeProcessed = requestedAmount % 100;
 if (pendingAmountToBeProcessed > 0)
 {
 rsHandler.dispatchRs(pendingAmountToBeProcessed);
 }
 }
}
```

FiftyRsHandler.java

```
public class FiftyRsHandler extends RsHandler
{
 public void dispatchRs(long requestedAmount)
```

```
{
 long numberofNotesToBeDispatched = requestedAmount / 50;
 if (numberofNotesToBeDispatched > 0)
 {
 if (numberofNotesToBeDispatched > 1)
 {
 System.out.println(numberofNotesToBeDispatched + " ,Fifty Rs notes are dispatched by
FiftyRsHandler\n");
 }
 else
 {
 System.out.println(numberofNotesToBeDispatched + " ,Fifty Rs note is dispatched by
FiftyRsHandler\n");
 }
 }
 long pendingAmountToBeProcessed = requestedAmount % 50;
 if (pendingAmountToBeProcessed > 0)
 {
 rsHandler.dispatchRs(pendingAmountToBeProcessed);
 }
 }
}
```

TwentyRsHandler.java

```
public class TwentyRsHandler extends RsHandler
{
 public void dispatchRs(long requestedAmount)
 {
```

```
 long numberofNotesToBeDispatched = requestedAmount / 20;
 if (numberofNotesToBeDispatched > 0)
 {
 if (numberofNotesToBeDispatched > 1)
 {
 System.out.println(numberofNotesToBeDispatched + " ,Twenty Rs notes are dispatched
by TwentyRsHandler\n");
 }
 else
 {
 System.out.println(numberofNotesToBeDispatched + " ,Twenty Rs note is dispatched by
TwentyRsHandler\n");
 }
 }
 long pendingAmountToBeProcessed = requestedAmount % 20;
 if (pendingAmountToBeProcessed > 0)
 {
 rsHandler.dispatchRs(pendingAmountToBeProcessed);
 }
 }         }
```

FiveRsHandler.java

```
public class FiveRsHandler extends RsHandler
{
 public void dispatchRs(long requestedAmount)
 {
 long numberofNotesToBeDispatched = requestedAmount / 5;
 if (numberofNotesToBeDispatched > 0)
```

```
{
if (numberofNotesToBeDispatched > 1)
{
System.out.println(numberofNotesToBeDispatched + " ,Five Rs notes are dispatched by FiveRsHandler\n");
}
else
{
System.out.println(numberofNotesToBeDispatched + " ,Five Rs note is dispatched by FiveRsHandler\n");
}
}
long pendingAmountToBeProcessed = requestedAmount % 5;
if (pendingAmountToBeProcessed > 0)
{
rsHandler.dispatchRs(pendingAmountToBeProcessed);
}
}
}
```

ATM.java

```
public class ATM
{
private static HundredRsHandler hundredRsHandler = new HundredRsHandler();
private static FiftyRsHandler   fiftyRsHandler   = new FiftyRsHandler();
private static TwentyRsHandler  twentyRsHandler  = new TwentyRsHandler();
private static FiveRsHandler    fiveRsHandler    = new FiveRsHandler();
static
```

```
 {
 // Prepare the chain of Handlers
 hundredRsHandler.nextHandler(fiftyRsHandler);
 fiftyRsHandler.nextHandler(twentyRsHandler);
 twentyRsHandler.nextHandler(fiveRsHandler);
 }
 public void withdraw( long requestedAmount )
 {
 hundredRsHandler.dispatchRs(requestedAmount);
 }
}
```

ChainOfResDemo.java

```
public class ChainOfResDemo
{
 public static void main( String[] args )
 {
 ATM atm = new ATM();
 System.out.println("\n ----------------Requested Amount 475----------------------\n");
 atm.withdraw(475);
 System.out.println("\n ----------------Requested Amount 455----------------------\n");
 atm.withdraw(455);
 System.out.println("\n ----------------Requested Amount 520----------------------\n");
 atm.withdraw(520);
 }
}
```

12.Output :

----------Requested Amount 475----------

4 ,Hundred Rs notes are dispatched by HundredRsHandler

1 ,Fifty Rs note is dispatched by FiftyRsHandler

1 ,Twenty Rs note is dispatched by TwentyRsHandler

1 ,Five Rs note is dispatched by FiveRsHandler

----------Requested Amount 455----------

4 ,Hundred Rs notes are dispatched by HundredRsHandler

1 ,Fifty Rs note is dispatched by FiftyRsHandler

1 ,Five Rs note is dispatched by FiveRsHandler

----------Requested Amount 520----------

5 ,Hundred Rs notes are dispatched by HundredRsHandler

1 ,Twenty Rs note is dispatched by TwentyRsHandler