



Хэрэглээний шинжлэх
ухаан, Инженерчлэлийн
сургууль

BEHAVIORAL DESIGN PATTERN



Намсрайдоржийн МӨНХЦЭЦЭГ

МЭДЭЭЛЭЛ, КОМПЬЮТЕРИЙН УХААНЫ ТЭНХИМ
МУИС, Хэрэглээний шинжлэх ухаан инженерчлэлийн сургууль
munkhtsetseg@seas.num.edu.mn



OVERVIEW

- Ашиглах ном
- Design Pattern-ы талаар
- Structural Design Patterns
- Behavioral Design Patterns

Books

- Design Patterns : Elements of Reusable Object-Oriented Software (1995)
 - (The-Gang-of-Four Book)
 - The-Gang-of-Four (GoF) - Gamma, Helm, Johnson , Vlissides
- Analysis Patterns - Reusable Object Models (1997)
 - Martin Fowler
- The Design Patterns Smalltalk Companion (1998)
 - Alpert, Brown & Woolf



DESIGN PATTERN-ЗОХИОМЖИЙН ЗАГВАР

“Зохиомжийн загвар бүр нь хүрээлэн буй орчинд дахин дахин гарч ирж буй асуудлыг тодорхойлдог бөгөөд үүний шийдлийн цөмийг тодорхойлж өгдөг. Ингэснээр энэ шийдлийг сая дахин давтахгүйгээр хоёр удаа хийж болно.”

--- Christopher Alexander, 1977

Паттерны асуудал нь бүх салбарт байх ба харин Програм хангамжийн инженерчлэлд объект болон интерфэйсуудын хооронд асуудал яригдана.

Тодорхой даалгаврыг биелүүлэх эсвэл тодорхой функцийг байгуулахын тулд харилцан үйлчлэлцэж буй объектуудын цуглуулга хэрхэн ажилладаг вэ гэдгийг тодорхойлж өгнө.

Architecture болон Design Pattern 2-н ялгаа

Architecture

- Программ хангамжийн бүтцийг тодорхойлсон дээд түвшний фреймворк
 - “client-server based on remote procedure calls”
 - “abstraction layering”
 - “distributed object-oriented system based on CORBA”
- Системийг бүрэлдүүлж байгаа компонентууд болон тэдгээрийн харилцааг харуулна.

Design Patterns

- Architecture-н доод түвшин (Заримдаа *micro-architecture* гэж нэрлэдэг)
- Аппликешн доторх дэд асуудлыг шийддэг дахин ашиглах боломжтой хамтын ажиллагаа.
 - Хэрхэн X дэд хэсгээс Y дэд хэсгийг салгах вэ?

Яагаад Design Patterns хэрэгтэй гэж?

- Design patterns нь хийсвэрлэлийн өндөр түвшинд объект руу чиглэсэн дахин ашиглалтыг дэмждэг.
- Design patterns нь объект хандлагыг хэрэгжүүлэх болон ашиглах гарын авлага болсон фреймворж юм.



DESIGN PATTERN-НЫ ҮНДСЭН 4 ЭЛЕМЕНТ

- *Нэр: Паттерн бүр нэртэй байх.*
- *Асуудал: Ямар асуудалд хэрэглэхийг тодорхой байх*
- *Шийдэл: Элементүүдийн хамтын ажиллагаа, харилцаа, үүрэг хариуцлагыг тодорхойлсон байх*
- *Үр дагавар: Паттернг хэрэглэх үр дүн тодорхой байх*



DESIGN PATTERN-Ы АНГИЛАЛ

- Зорилгын хувьд:
 - Creational Patterns
 - Structural Patterns
 - Behavioral Patterns



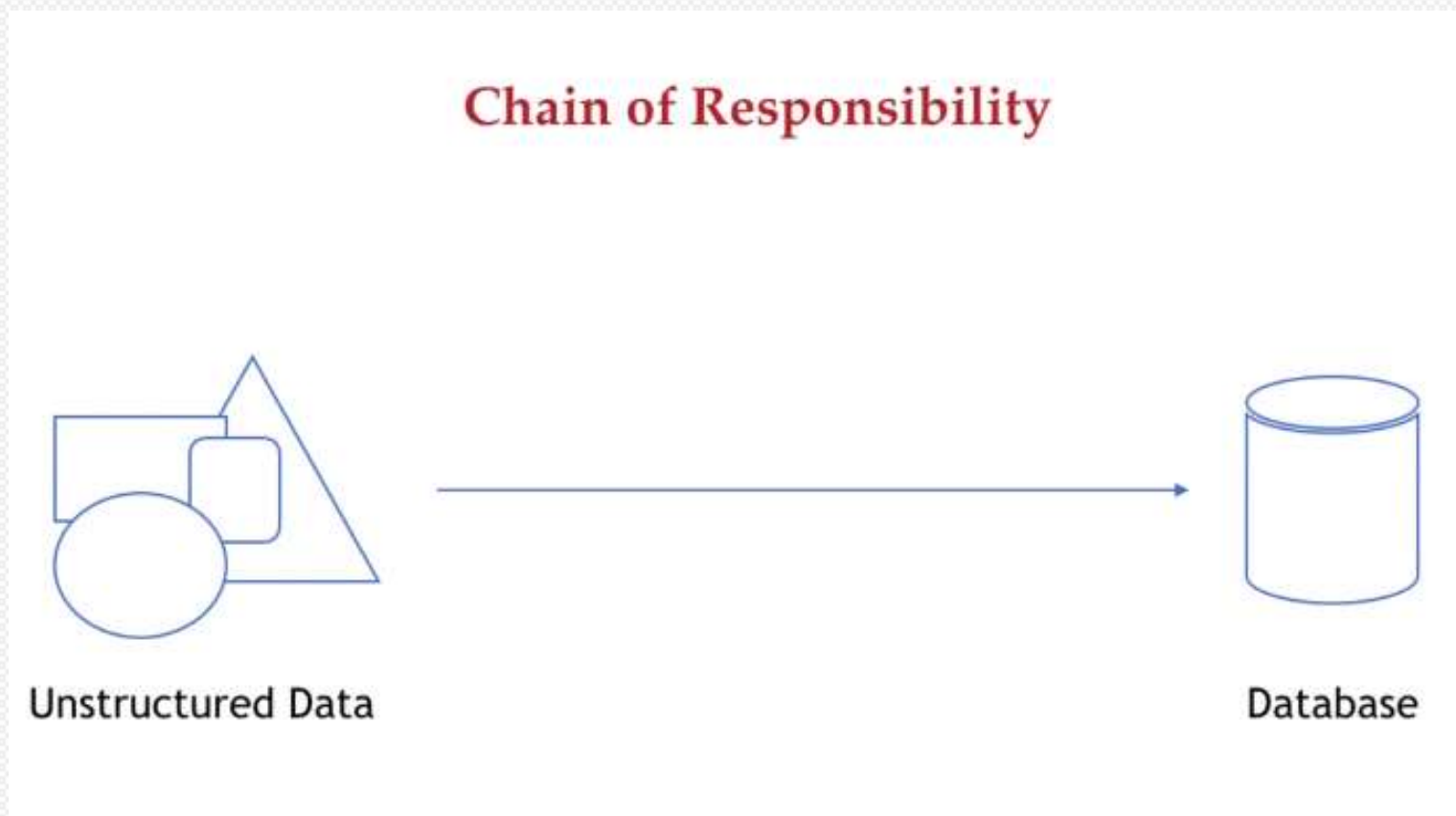
BEHAVIORAL PATTERNS

- Behavioral patterns нь объектуудыг хоорондын харилцаа ба үүрэг хариуцлагыг аль болохоор бие биенээс нь бага хамааралтай байхаар (loose coupled) холбодог.
- Chain of Responsibility Design Pattern
- State Design Pattern
- Mediator Design Pattern
- Observer Design Pattern
- Memento Design Pattern
- Iterator Design Pattern
- Command Design Pattern
- Strategy Design Pattern
- Template Method Design Pattern
- Visitor Design Pattern Example



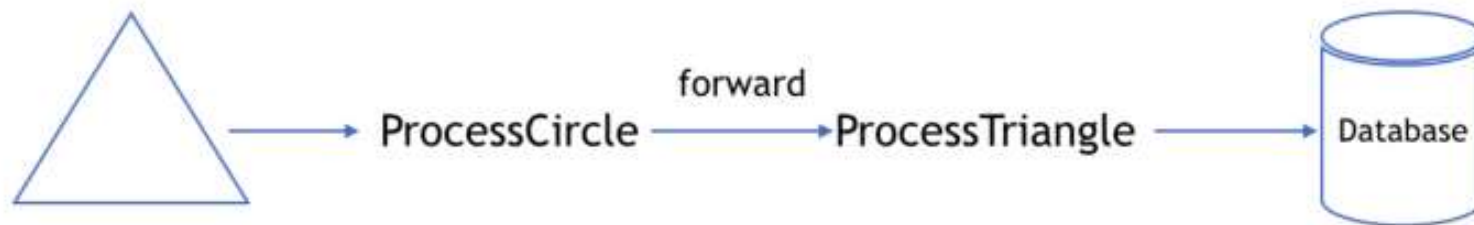
CHAIN OF RESPONSIBILITY DESIGN PATTERN

- Гинжин объектыг хүсэлтийг хүлээн авахад хэрэглэнэ.
- Тухайн үед үйлчилж байгаа хүсэлтийн объектоос хэрэглэгчийг салгаж өгнө. Тухайн хэрэглэгч аль объект нь түүний хүсэлтэнд үйлчилж байгааг мэдэхгүй байдаг.



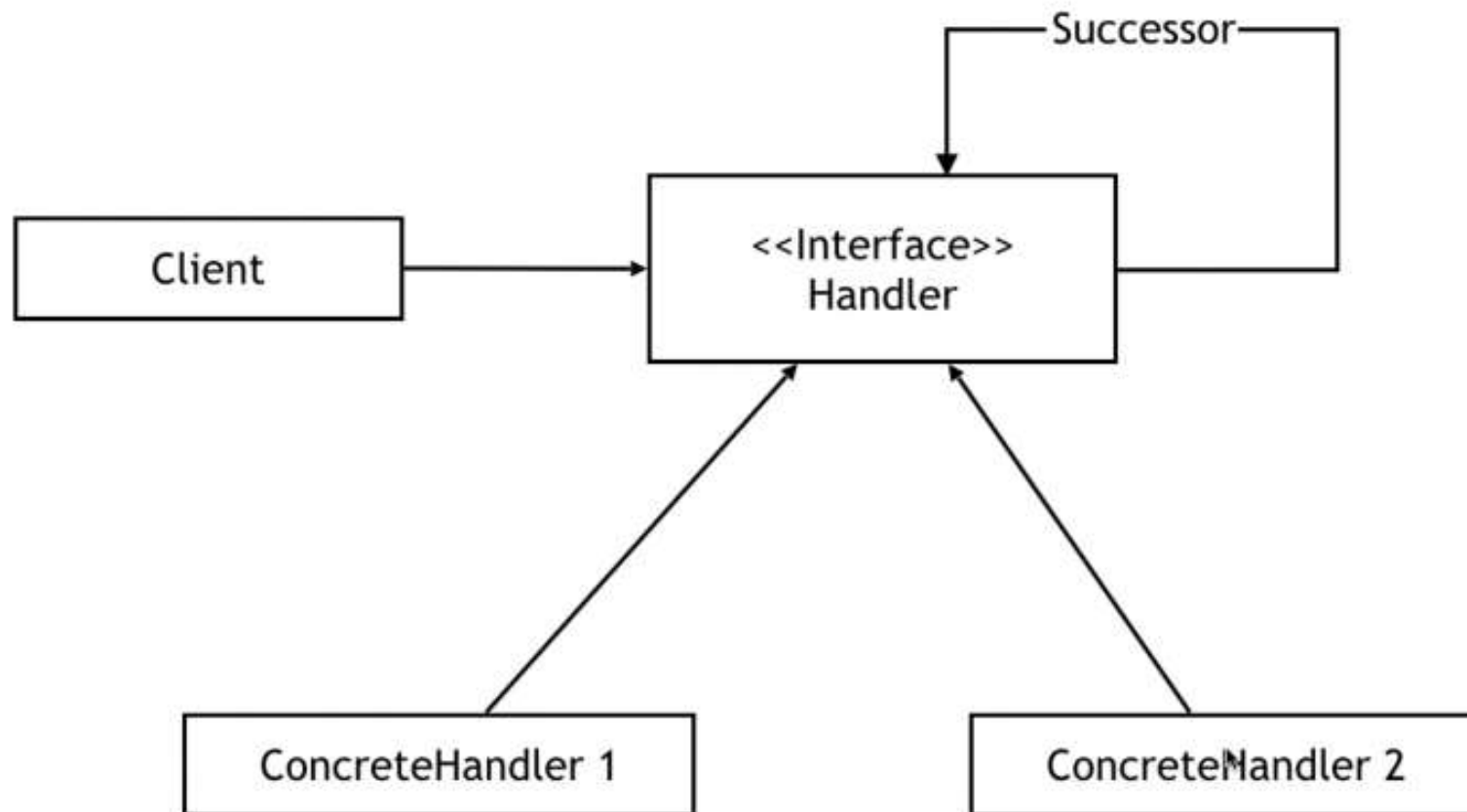


Chain of Responsibility



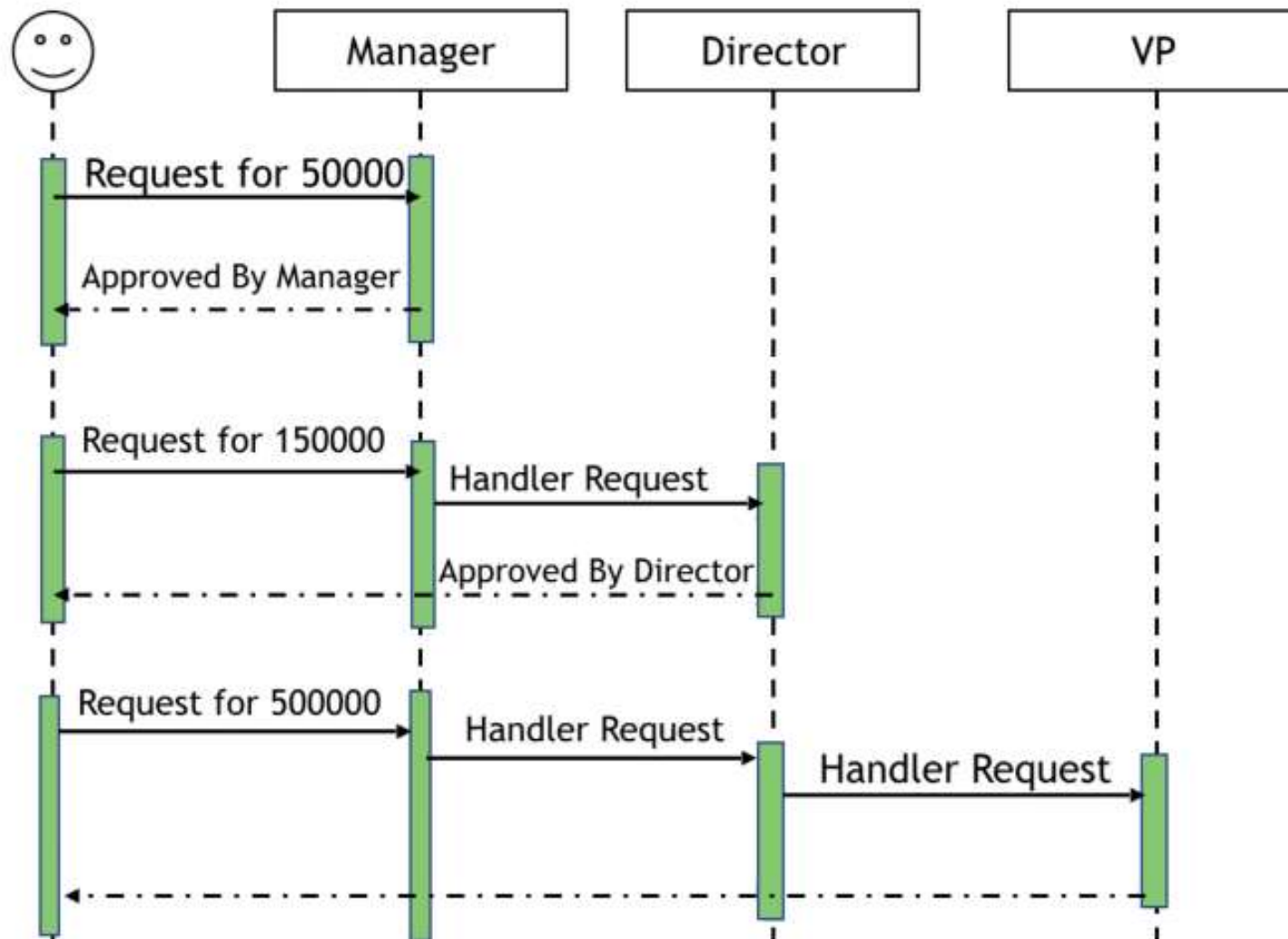


UML Class Diagram of Chain of Responsibility





UML Sequence Diagram of Chain of Responsibility





```
class Loan {  
  
    private int amount;  
  
    public int getAmount() {  
        return amount;  
    }  
  
    public Loan(int amount) {  
        this.amount = amount;  
    }  
}
```

```
abstract class LoanApproval{  
  
    protected LoanApproval  
        nextLoanApproval;  
  
    public void  
        setNextLoanApproval(LoanApproval  
            nextLoanApproval) {  
        this.nextLoanApproval=nextLoanAppr  
            oval;  
    }  
  
    public abstract void  
        approveLoan(Loan loan);  
}
```



```
class Director extends LoanApproval{
public void approveLoan(Loan loan){
if (loan.getAmount()<=250000) {
System.out.println("Захирагч зөвшөөрсөн");
}

else
nextLoanApproval.approveLoan(loan) ;
}
}
```

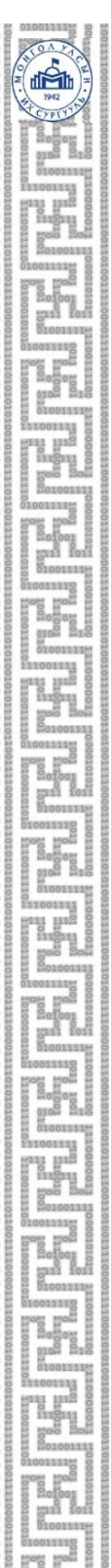
```
class Manager extends LoanApproval{

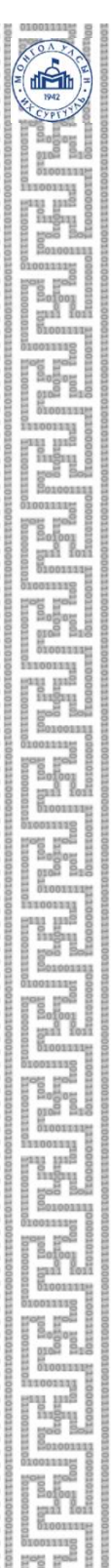
public void approveLoan(Loan loan){
if (loan.getAmount()<=100000) {
System.out.println("Менежер зөвшөөрсөн");
}
else
nextLoanApproval.approveLoan(loan);
}
}
```

```
class VicePresident extends
LoanApproval{
public void approveLoan(Loan
loan){
System.out.println("Дэд
ерөнхийлөгч зөвшөөрсөн");
} }-
```




```
public class ChainOf {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        LoanApproval m=new Manager();  
        LoanApproval d=new Director();  
        LoanApproval vp=new VicePresident();  
        m.setNextLoanApproval(d);  
        d.setNextLoanApproval(vp);  
        m.approveLoan(new Loan(500000));  
    }  
}
```







When to use

- The chain of responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle a request.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

Chain of Responsibility in JDK

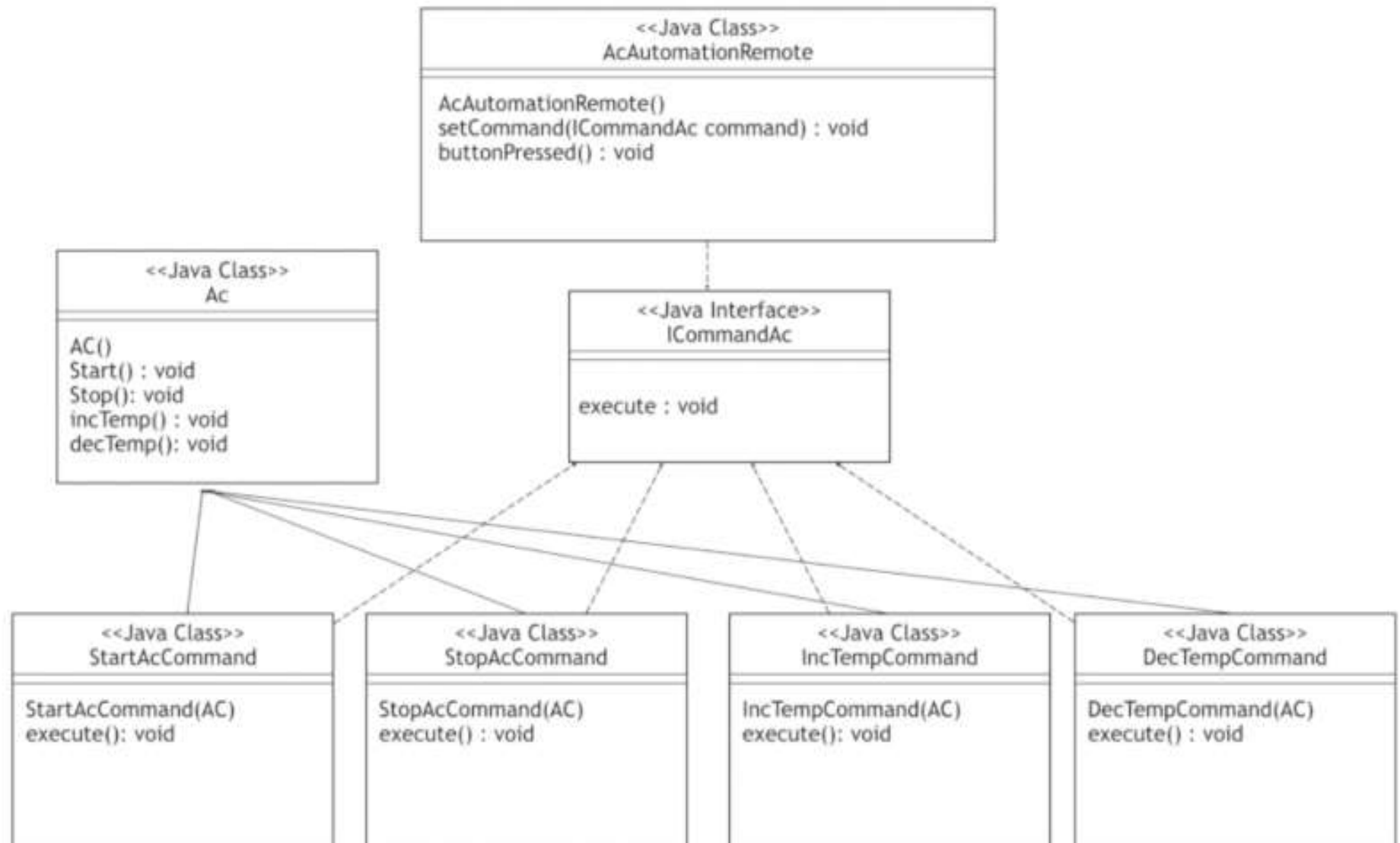
- `java.util.logging.Logger#log()`
- `javax.servlet.Filter#doFilter()`



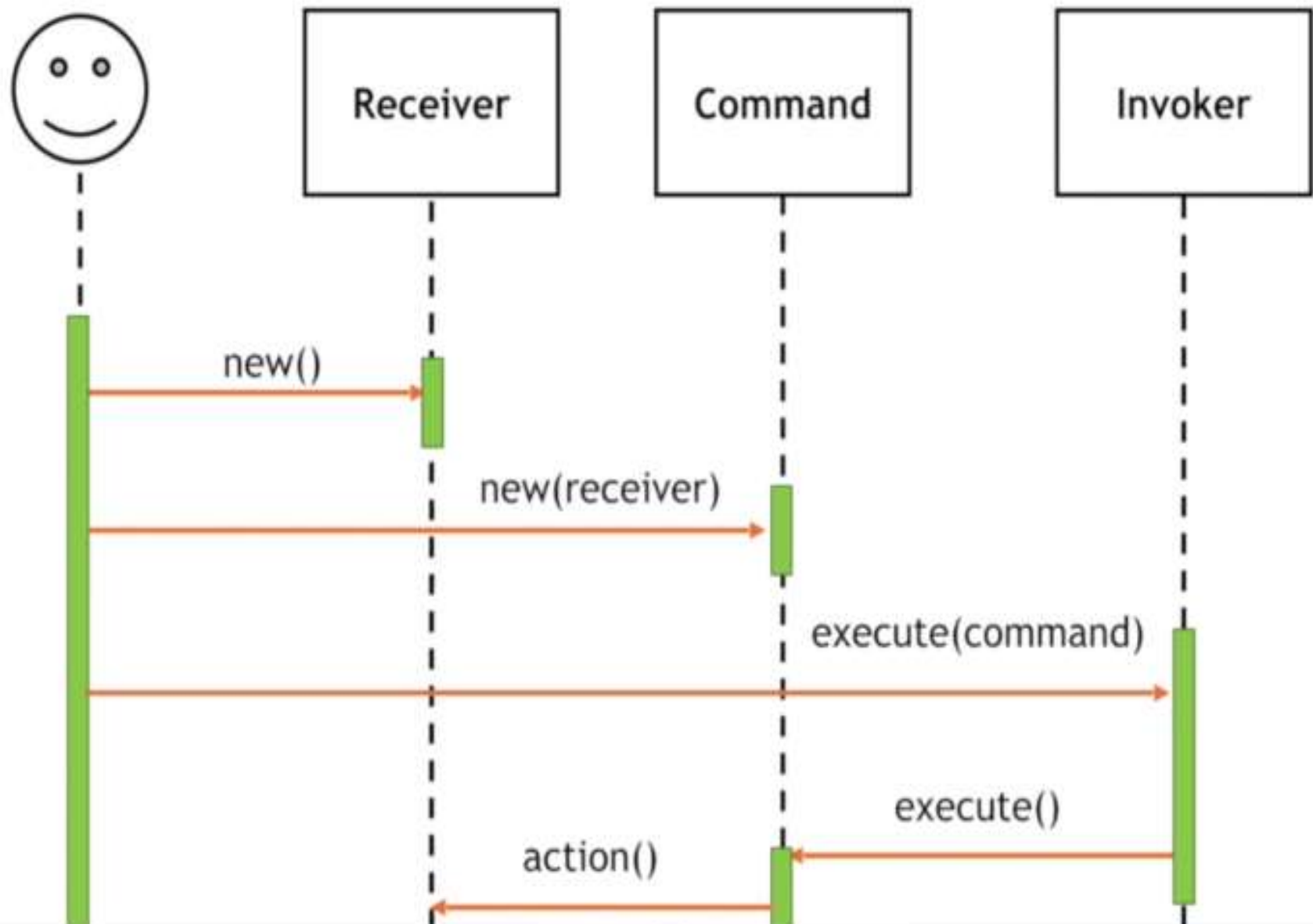
COMMAND DESIGN PATTERN

- Ямар нэгэн зүйлийн объект өөрийн property-ууд болон дүрмүүдийг бүгдийг агуулна. Харин түүний өөр өөр объектууд нь дүрмүүдийг өөр өөрөөр хэрэгжүүлэх боломжийг олгоно.
- Дараах ойлголтууд байна. Үүнд:
 - Комманд interface
 - Бодит комманд класс
 - Invoker буюу дуудагч
 - Receiver буюу хүлээн авагч

UML Class Diagram of Command Design Pattern



UML Sequence Diagram of Command Design Pattern





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

```
class AC{  
public void turnOn() {  
System.out.println("Үндсэн AC аслаа");  
}
```

```
public void turnOff() {  
System.out.println("Үндсэн AC унтарлаа ");  
}
```

```
public void decTemp() {  
System.out.println("Үндсэн AC Температурыг  
багасгалаа.");  
}
```

```
public void incTemp() {  
System.out.println("Үндсэн AC Температурыг  
ихэсгэлээ.");  
}  
}
```




```
class IncTempACCommand implements  
ICommandAC{
```

```
    AC ac;  
    public IncTempACCommand(AC ac) {  
        this.ac=ac;  
    }  
    @Override  
    public void execute() {  
        System.out.println(" execute  
ажиллаж AC -н температур  
нэмэгдлээ");  
        ac.incTemp();  
    }  
}e
```

```
class StartACCommand implements  
ICommandAC{
```

```
    AC ac;  
    public StartACCommand(AC ac) {  
        this.ac=ac;  
    }  
    @Override  
    public void execute() {  
        System.out.println("execute  
ажиллаж AC аслаа ");  
        ac.turnOn();  
    }  
}
```




```
class DecTempACCommand implements  
ICommandAC{
```

```
    AC ac;  
    public DecTempACCommand(AC ac) {  
        this.ac=ac;  
    }  
    @Override  
    public void execute() {  
        System.out.println("execute  
ажиллаж AC -н температур  
буурлаа");  
        ac.decTemp();  
    }  
}
```

```
class StopACCommand implements  
ICommandAC{
```

```
    AC ac;  
    public StopACCommand(AC ac) {  
        this.ac=ac;  
    }  
    @Override  
    public void execute() {  
        System.out.println("execute
```

Invoker
классе

```
class ACAutomationRemote{
    ICommandAC command;

    public void setCommand(ICommandAC
    command) {
        this.command=command;
    }

    public void buttonPressed() {
    } command.execute();

    }
}
```

```
public class CommandDesignPattern
{
    public static void main(String[]
args) {
        AC acRoomOne = new AC();
        AC acRoomTwo = new AC();

        ACAutomationRemote remote = new
        ACAutomationRemote();
        remote.setCommand(new
        StartACCommand(acRoomOne));
        remote.buttonPressed();
        remote.setCommand(new
        StartACCommand(acRoomTwo));
        remote.buttonPressed();
        remote.setCommand(new
        IncTempACCommand(acRoomTwo));
        remote.buttonPressed();
    }
}
```



When to use

- You want to be able to parameterise objects by an action to perform.
- You need to specify, queue, and execute requests at different times.
- When a set of **changes to data** need to be encapsulated as a single action (i.e. a transaction).

Command Design pattern in JDK

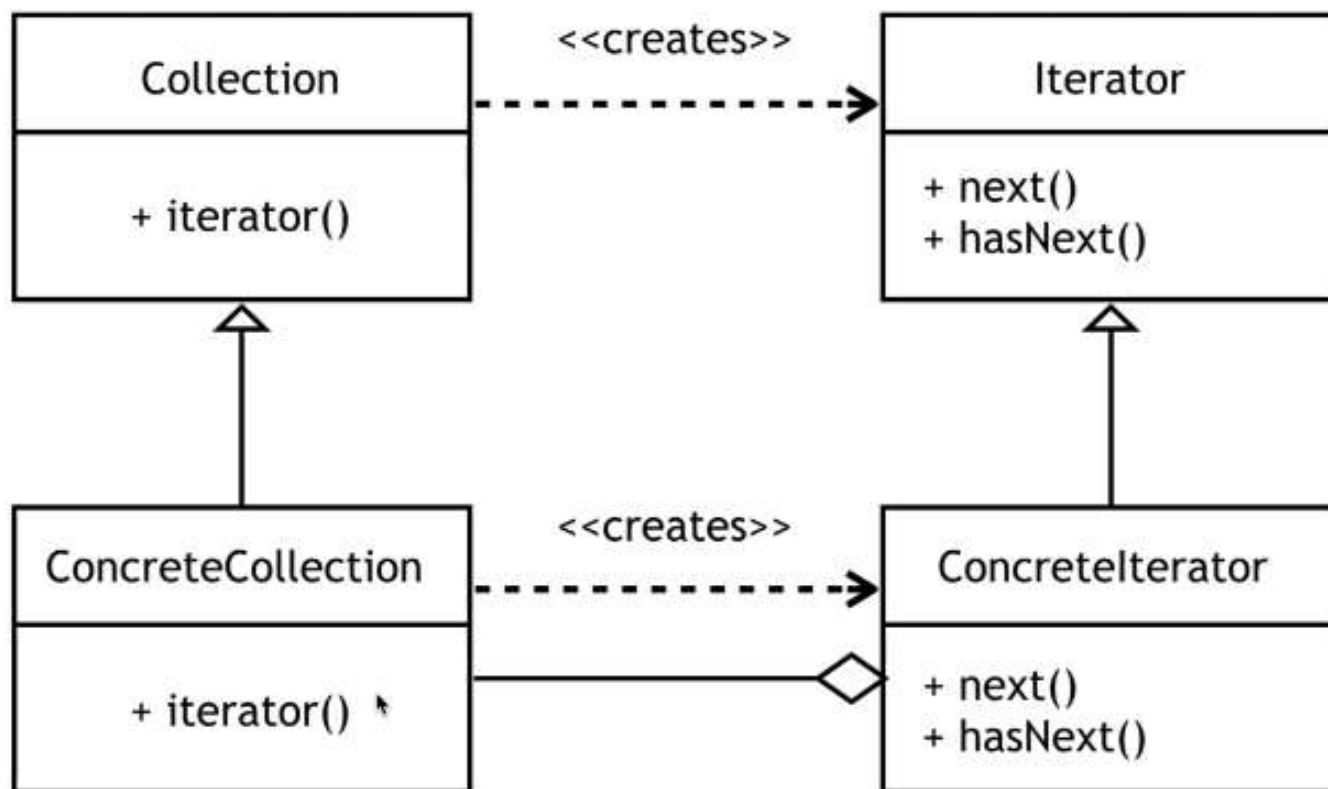
- `java.lang.Runnable`
- `javax.swing.Action`



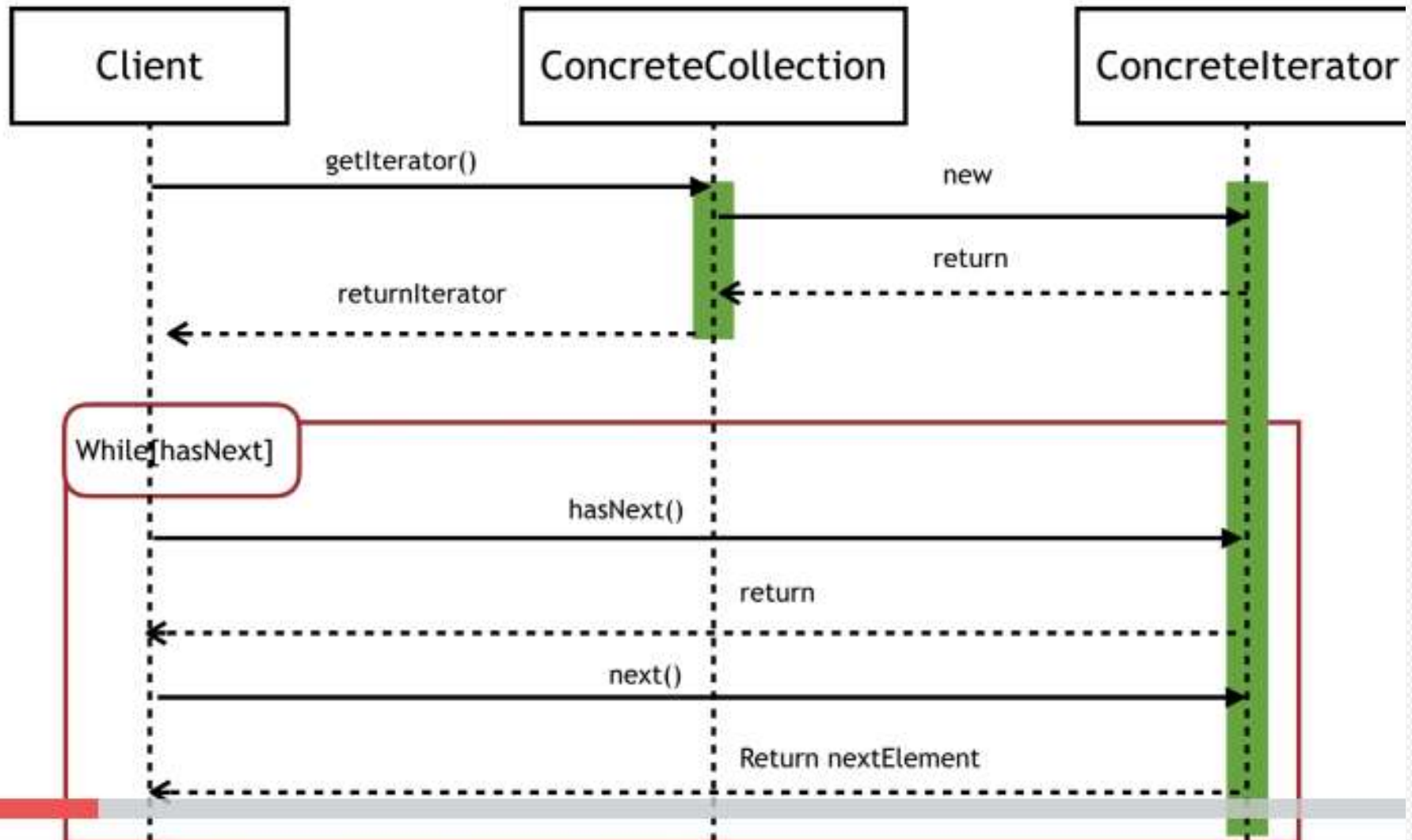
ITERATOR DESIGN PATTERN

- Дотоод бүтцийг нь мэдэхгүй дараалсан элемент рүү хандаж өгөгдөл авах боломжтой.

UML class diagram of Iterator Design Pattern



UML Sequence diagram of Iterator Design Pattern





```
package Iterator;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Set;
public class IteratorDesignPattern {
public static void main(String[] args) {
// TODO Auto-generated method stub
List<String> list=new ArrayList<String>();
list.add("Бат");
list.add("Дорж");
list.add("Долгор");
list.add("Наран");

Iterator<String> itr=list.iterator();

while(itr.hasNext()) {
System.out.println("Нэр :" + itr.next());
}
System.out.println("Бодит жагсаалт:
"+itr.toString());
```

```
Set<String> set=new HashSet<String>();

set.add("Саран");
set.add("Дэлгэр");
set.add("Саруул");

Iterator<String>
setItr=set.iterator();
while(setItr.hasNext()) {
System.out.println("Нэр :
"+setItr.next());
}}}
```



When to use

- To access an aggregate object's contents without exposing its internal representation.
- To support multiple traversals of aggregate objects.
- To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

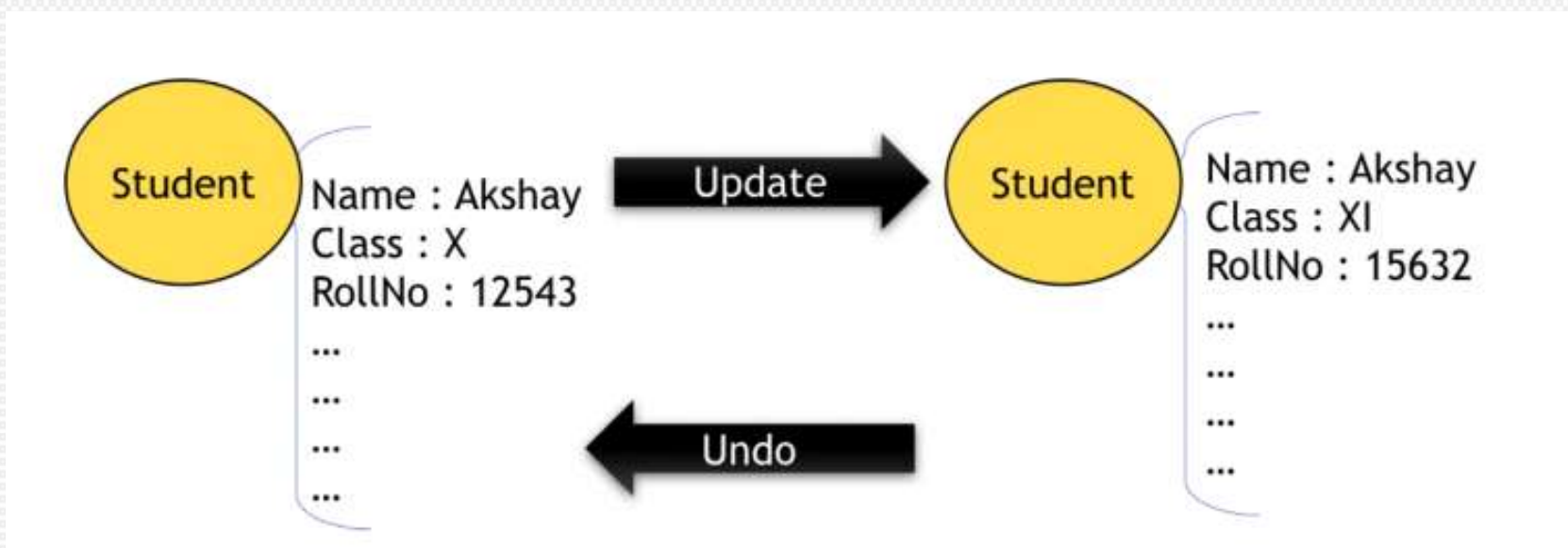
Iterator Design Pattern in JDK

- `java.util.Iterator`
- `java.util.Enumeration`



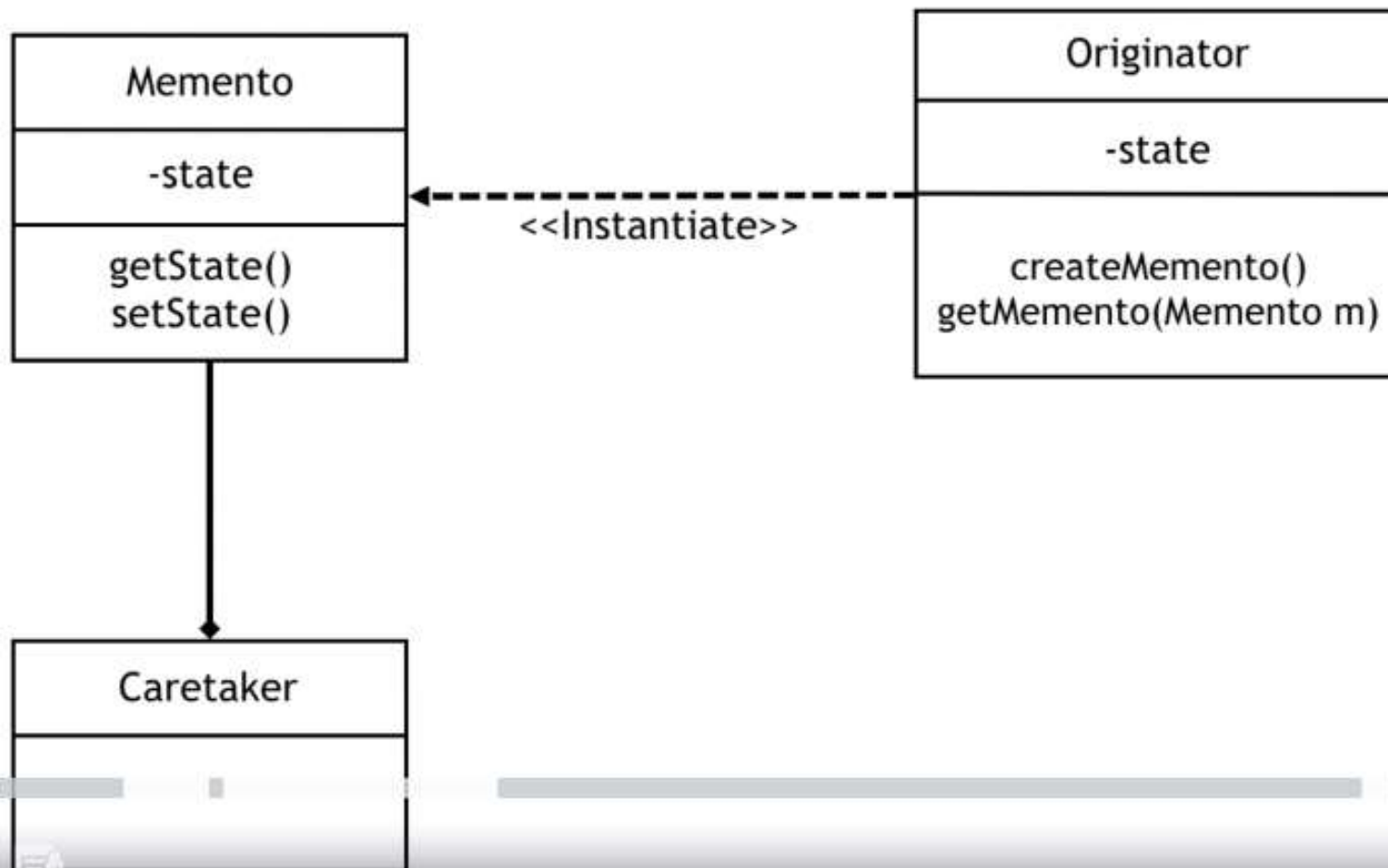
MEMENTO DESIGN PATTERN

- Объектын өмнөх төлвийг буцаана.



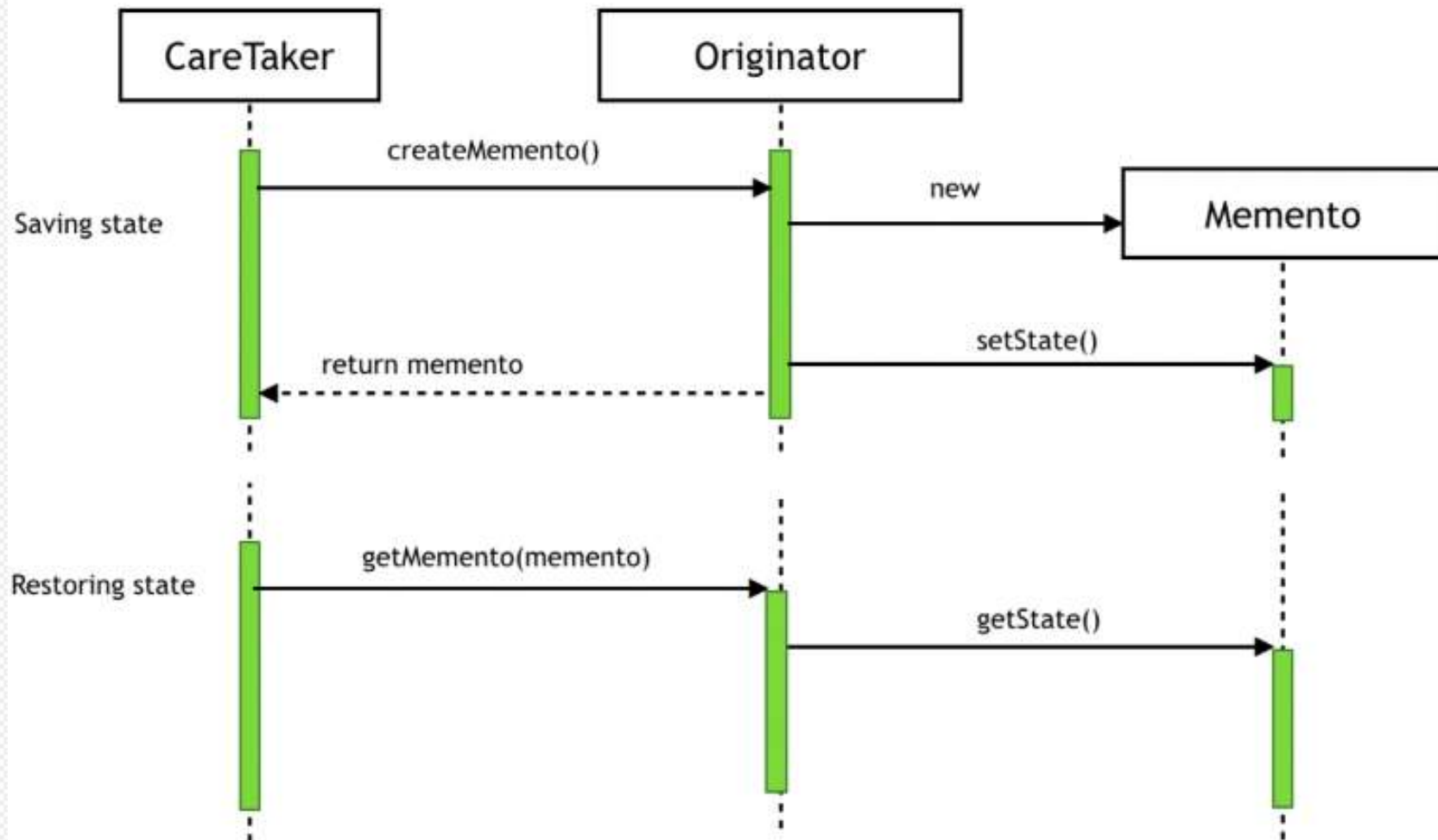


UML Class Diagram of Memento Design Pattern





UML Sequence Diagram of Memento Design Pattern





```
class Refrigerator{
private int price;
private int volume;
private boolean isPowerSaver;

public int getPrice() {
return price;
}

public void setPrice(int price) {
this.price = price;
}

public int getVolume() {
return volume;
}

public void setVolume(int volume) {
this.volume = volume;
}

public boolean isPowerSaver() {
return isPowerSaver;
}

public void setPowerSaver(boolean isPowerSaver) {
this.isPowerSaver = isPowerSaver;
}

public Refrigerator(int price, int volume, boolean isPowerSaver) {
super();
this.price=price;
this.volume=volume;
this.isPowerSaver=isPowerSaver;
}

@Override
public String toString() {
return "Refrigerator [price=" + price + ", volume=" + volume + ", isPowerSaver=" + isPowerSaver + "]";
}
```


```
class Memento{
Refrigerator refri;
```

```
public Refrigerator getRefri() {
return refri;
}
```

```
public void setRefri(Refrigerator
refri) {
this.refri = refri;
}
```

```
public Memento(Refrigerator refri)
{
this.refri=refri;
}
}
```

```
@Override
public String toString() {
return "Memento [refri=" + refri
+"]";
}
}
```

```
class LivingAreaOriginator{
    Refrigerator ref;

    public Refrigerator getRef() {
        return ref;
    }

    public void setRef(Refrigerator ref) {
        this.ref = ref;
    }

    public Memento createMemento() {
        return new Memento(ref);
    }

    public void getMemento(Memento m) {
        ref=m.getRefri();
    }

    @Override
    public String toString() {
        return "LivingAreaOriginator [ref=" + ref +
            "]";
    }
}
```

```
class CareTakerStore{
    List<Memento> refLists=new
    ArrayList<Memento>();

    public void addMemento(Memento
    memento) {
        refLists.add(memento);
    }

    public Memento getMemento(int
    index) {
        return refLists.get(index);
    }
}
```



```
public class MementoDesignPattern {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        LivingAreaOriginator originator=new
        LivingAreaOriginator();
        CareTakerStore ct=new CareTakerStore();

        originator.setRef(new Refrigerator(15000,15,false));
        ct.addMemento(originator.createMemento());

        originator.setRef(new Refrigerator(25000,20,true));
        ct.addMemento(originator.createMemento());

        System.out.println("Одоогийн байгаа төлвийг харуулж
        байна.");
        System.out.println("originator-н одоогийн
        төлөв"+originator);
        System.out.println("Өмнөхийг байсан төлвийг харуулж
        байна.");

        originator.getMemento(ct.getMemento(0));

        System.out.println("Өмнөх төлөв :"+originator);

    }

}
```




When to use...

- Some portion of an object's state must be saved so that it can be restored to that state later.
- A direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

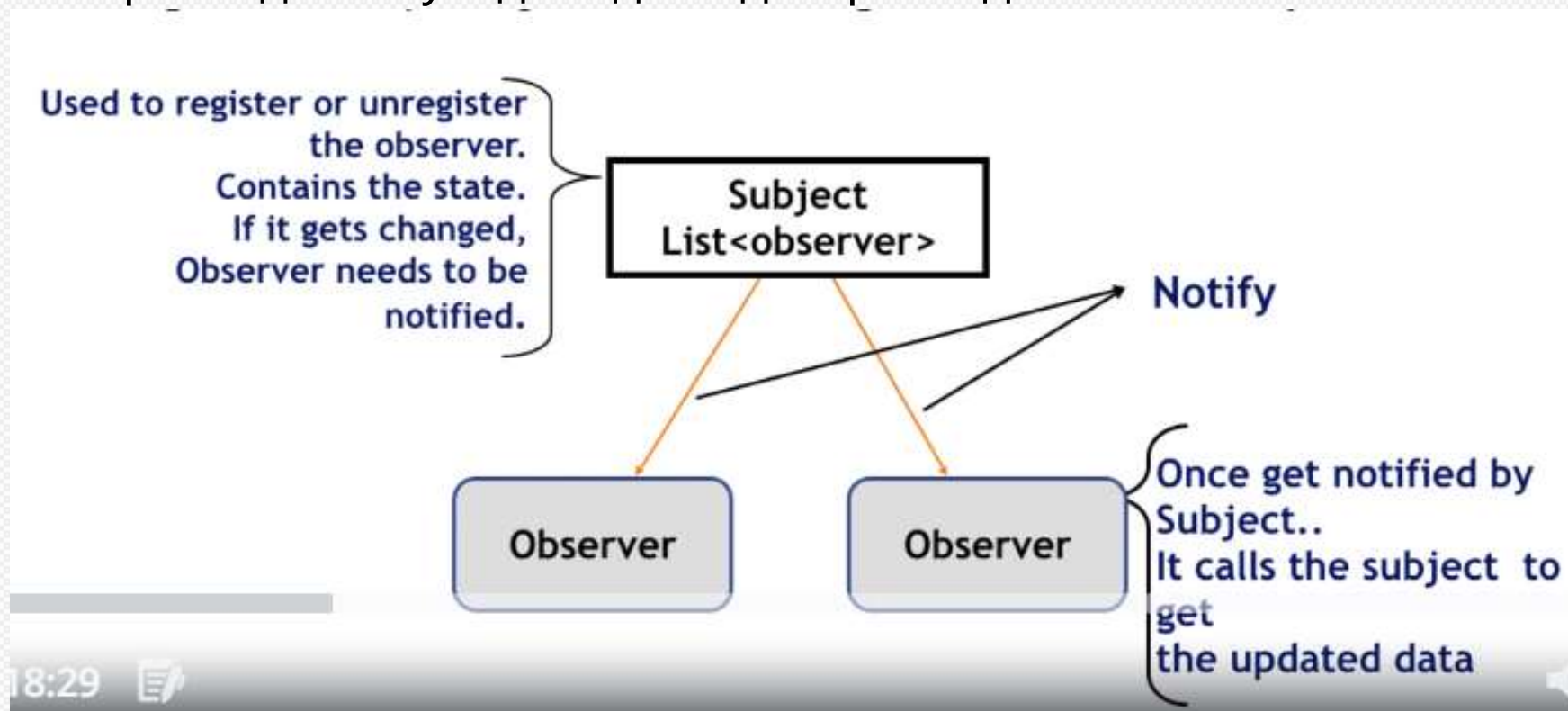
Memento Pattern in JDK

- `java.util.Date`
- `java.io.Serializable`



OBSERVER DESIGN PATTERN

- Нэг объект олон объектуудаас хамаарна. Нэг объектын төлөв өөрчлөгдвөл бусад нь дагаад өөрчлөгдөнө.



18:29



Real Time Example of Observer Design Pattern



Select the notify me option.

Real Time Example of Observer Design Pattern



Select the notify me option.



Notify

Notify

Notify



Bob



Rob



Rooney

Observers

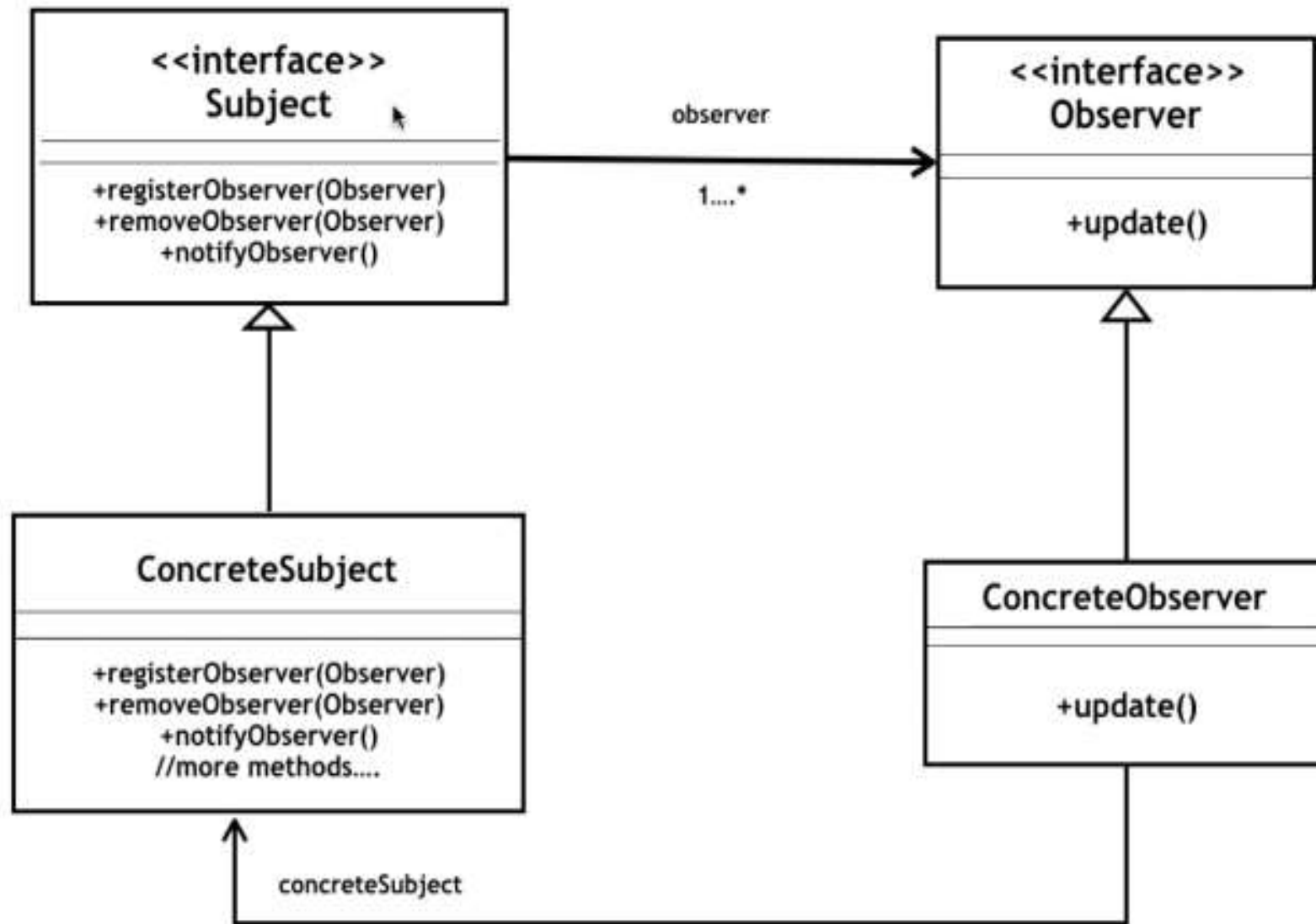


Not Available

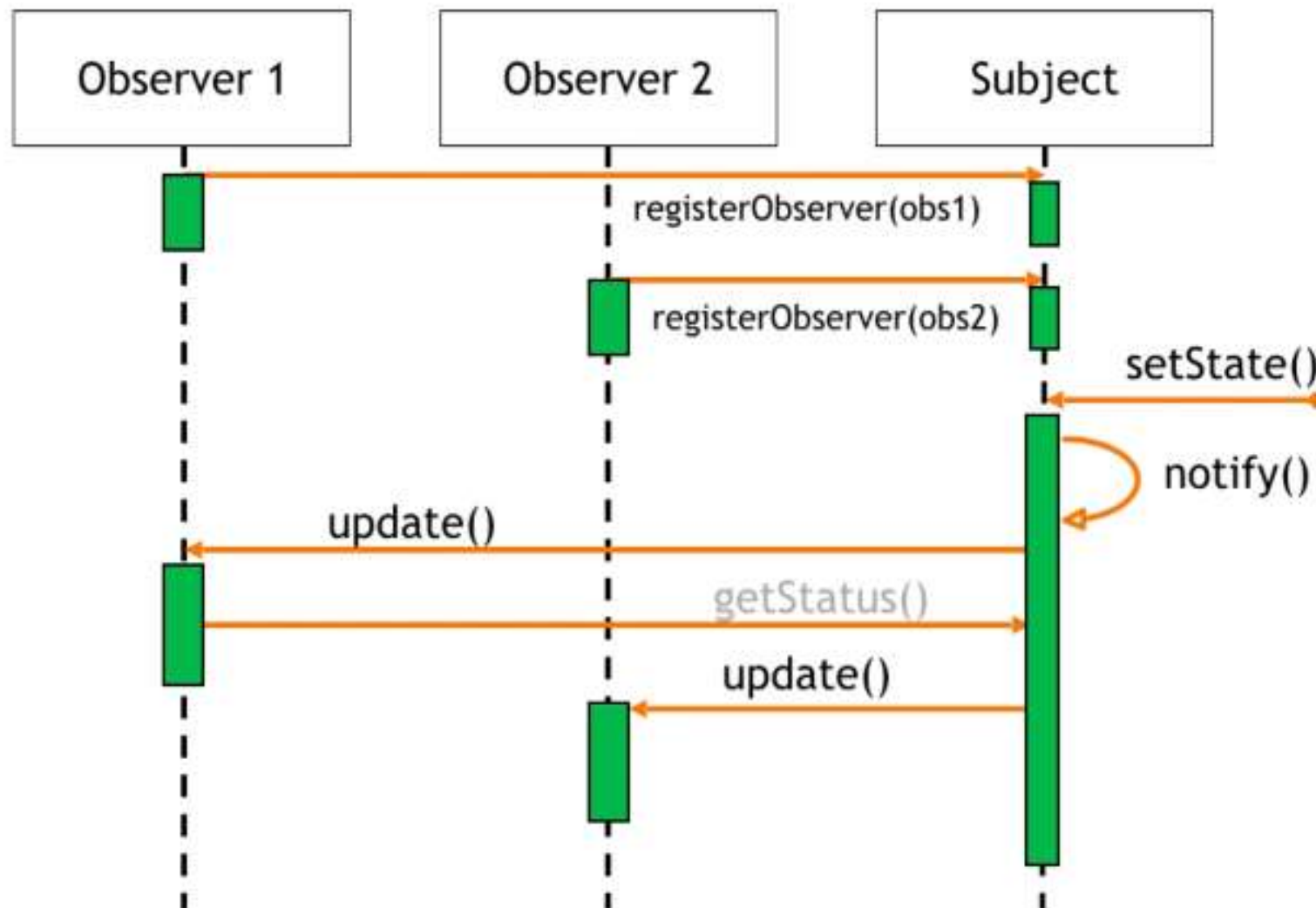


In Stock

UML Class Diagram of Observer Design Pattern



UML Sequence Diagram of Observer Design Pattern



```

import java.util.ArrayList;
public class Book implements SubjectLibrary {
    private String name;
    private String type;
    private String author;
    private double price;
    private String inStock;
    private ArrayList<Observer> obsList=new ArrayList<Observer>();

    public Book(String name, String type, String author, double price, String inStock) {
        this.name=name;
        this.type=type;
        this.author=author;
        this.price=price;
        this.inStock=inStock;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String getInStock() {
        return inStock;
    }

    public void setInStock(String inStock) {
        this.inStock = inStock;
    }
}

```

```
package PackageObserver;
```

```
public interface SubjectLibrary {
```

```
    public void
```

```
    subscribeObserver(Observer ob);
```

```
    public void
```

```
    unsubscribeObserver(Observer ob);
```

```
    public void notifyObserver();
```

```
}
```

```
package PackageObserver;
```

```
public interface Observer {
```

```
    public void update(String avail);
```

```
}
```




```
package PackageObserver;

public class EndUser implements Observer {
    String name;
    EndUser(String name, SubjectLibrary subject){
        this.name=name;
        subject.subscribeObserver(this);
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public void update(String avail) {
        // TODO Auto-generated method stub
        System.out.println("Сайн байна уу " + name + " бид  
мэдээлэл авсандаа баяртай байна. "+ avail);
    }
}
```



```
package PackageObserver;
```

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

```
import java.util.Iterator;
```

```
public class ObserverDesignPattern {
```

```
    public static void main(String[] args) {
```

```
        // TODO Auto-generated method stub
```

```
        Book book=new Book("Алгоритм  
програмчлал","Компьютер","Д.Гармаа", 20000,"Зарагдсан");
```

```
        EndUser user1=new EndUser("Мөнхцэцэг", book);
```

```
        EndUser user2=new EndUser("Бат", book);
```

```
        ArrayList<Observer> subscribers=book.getObsList();
```

```
        for(Iterator<Observer> itr=subscribers.iterator();
```

```
itr.hasNext();) {
```

```
            EndUser eu=(EndUser) itr.next();
```

```
            System.out.println(eu+" Энэ номонд захиалга  
өгсөн байна. " + book.getName()+ " Ном");
```

```
        }
```

```
        System.out.println(book.getlnStock());
```

```
        System.out.println(" Ном бэлэн байна.");
```

```
        book.setlnStock("Ном буцаагдаж ирсэн байна.");
```

```
    }
```

```
}
```




TEMPLATE DESIGN PATTERN

- Template design pattern нь суперкласс дахь дүрмүүдийг хийсвэр байдлаар тодорхойлоод түүнийг дэд классуудад өөр өөрийн онцлогоор хэрэгжүүлнэ.





Real Time Example of Template Design Pattern

Order and Payment Processing



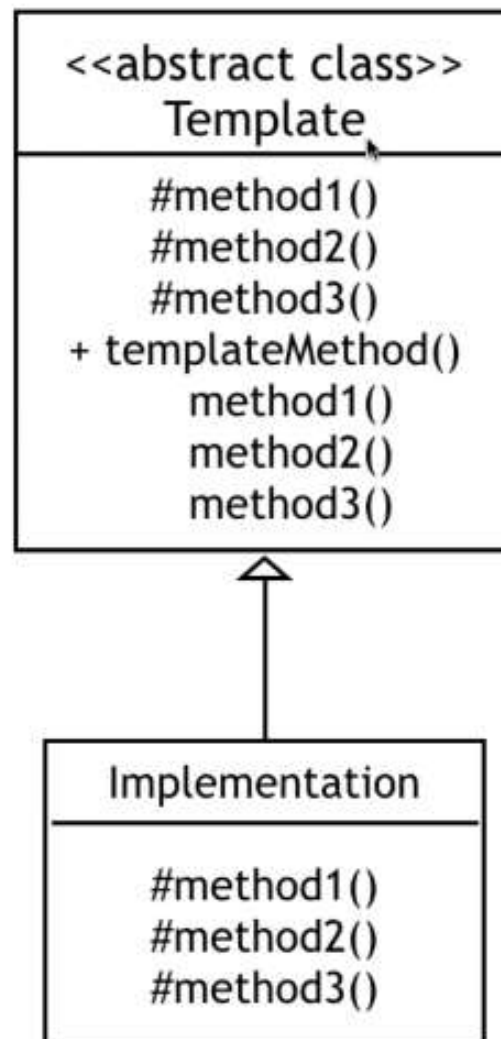
Shopping Mall



Online Shopping

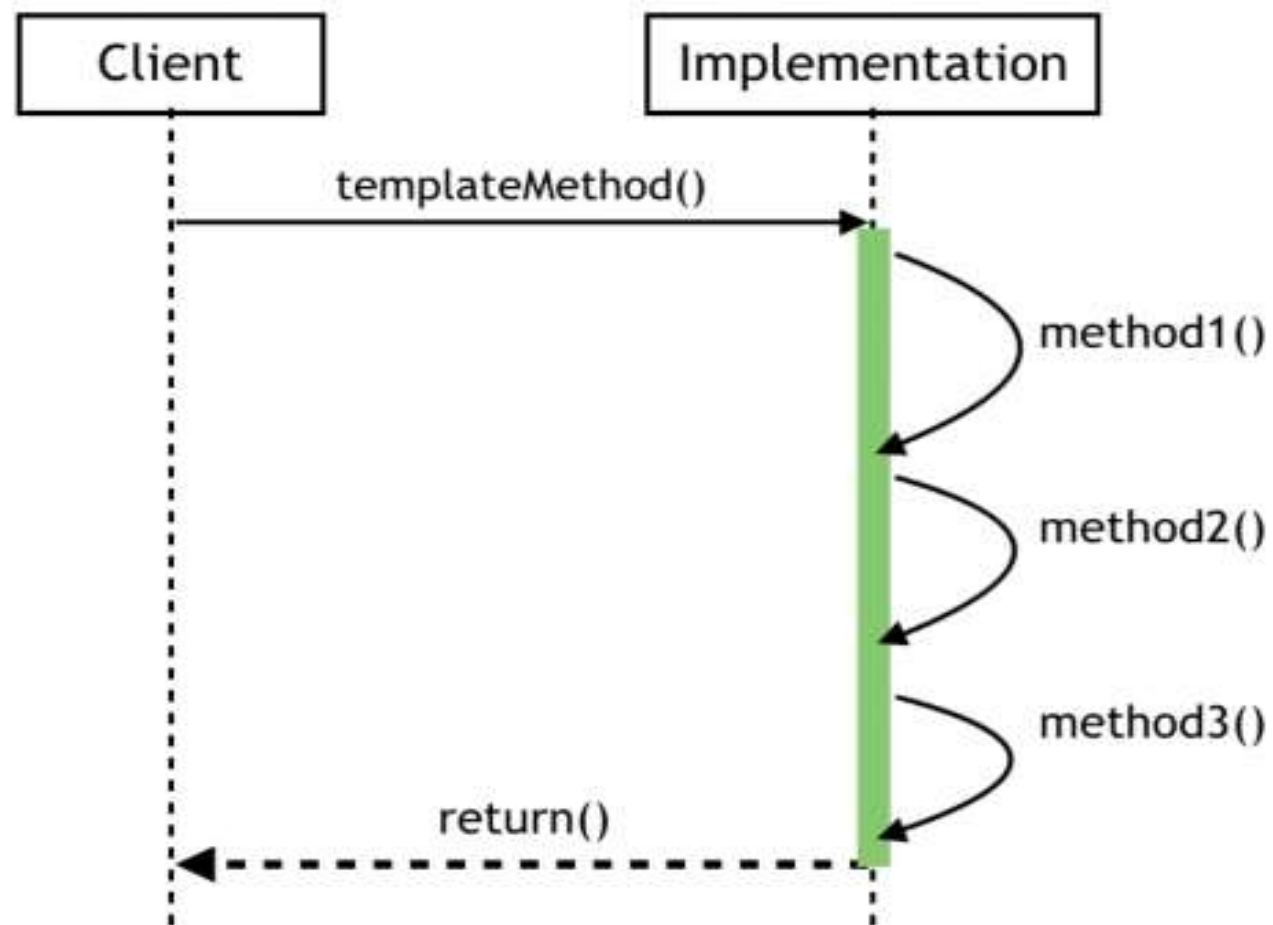


UML Class Diagram Of Strategy Design Pattern





UML Sequence Diagram Of Strategy Design Pattern





```
package Packagetemplate;
```

```
public abstract class ProcessOrder
{
    abstract public void
    selectProduct();
    abstract public void
    makePayment();
    abstract public void deliver();

    public final void doShopping() {
        selectProduct();
        makePayment();
        deliver();
    }
}
```

```
package Packagetemplate;
```

```
public class Flipkart extends ProcessOrder
{
    @Override
    public void selectProduct() {
        // TODO Auto-generated method stub
        System.out.println(" Онлайн худалдаа ");
        System.out.println(" Хэрэгтэй бараа ");
        System.out.println(" Карт руугаа нэмэх ");
        System.out.println(" Төлбөрөө хийх ");
    }
    @Override
    public void makePayment() {
        // TODO Auto-generated method stub
        System.out.println(" Хаягаа оруулна ");
        System.out.println(" Төлбөрийн хэлбэр ");
        System.out.println(" Төлбөрөө хийлэ ");
    }
    @Override
    public void deliver() {
        // TODO Auto-generated method stub
        System.out.println(" Таны бараа 3-4 ");
        System.out.println(" хүргэгдэнэ.");
        System.out.println(" Ахиж худалдан ");
        System.out.println(" ----- ");
    }
}
```



```
package Packagetemplate;
```

```
public class Shop extends ProcessOrder {
    @Override
    public void selectProduct() {
        // TODO Auto-generated method stub
        System.out.println(" Оффлайн худалдаа");
        System.out.println(" Хэрэгтэй бараагаа хайж ба

    }
    @Override
    public void makePayment() {
        // TODO Auto-generated method stub
        System.out.println(" Касс руу явлаа..");
        System.out.println(" Дараалалд зогслоо.");
        System.out.println(" Төлбөрөө хийлээ.");
        System.out.println(" Төлбөр төлсөн баримтаа аб

    }
    @Override
    public void deliver() {
        // TODO Auto-generated method stub
        System.out.println(" Төлбөр төлсөн баримтаа үзэ
        System.out.println(" Бараагаа олгох тасгаас аб
        System.out.println(" Манайхаар үйлчлүүлсэнд
        баярлалаа.");
        System.out.println(" -----");
    }
}
```

```
package Packagetemplate;
```

```
public class TempaleDesign
{
    public static void main(Str
    args) {
        // TODO Auto-generated meth
        ProcessOrder shopVisit =new
        Shop();
        shopVisit.doShopping();

        ProcessOrder onlineShopping
        Flipkart();
        onlineShopping.doShopping()

    }
}
```

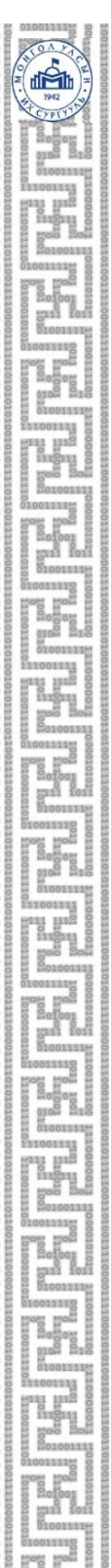



When to use

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behaviour that can vary.
- When common behaviour among subclasses should be factored and localised in a common class to avoid code duplication. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.

Template Design Pattern in Java

- `java.util.Collections#sort()`
- `java.io.InputStream#skip()`
- `java.io.InputStream#read()`
- `java.util.AbstractList#indexOf()`



АНХААРАЛ ХАНДУУЛСАНД БАЯРЛАЛАА.
