Хэрэглээний шинжлэх
ухаан, Инженерчлэлийн
сургууль

STRUCTURAL DESIGN PATTERN

Намсрайдоржийн МӨНХЦЭЦЭГ

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

Ерөнхий агуулга

- 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

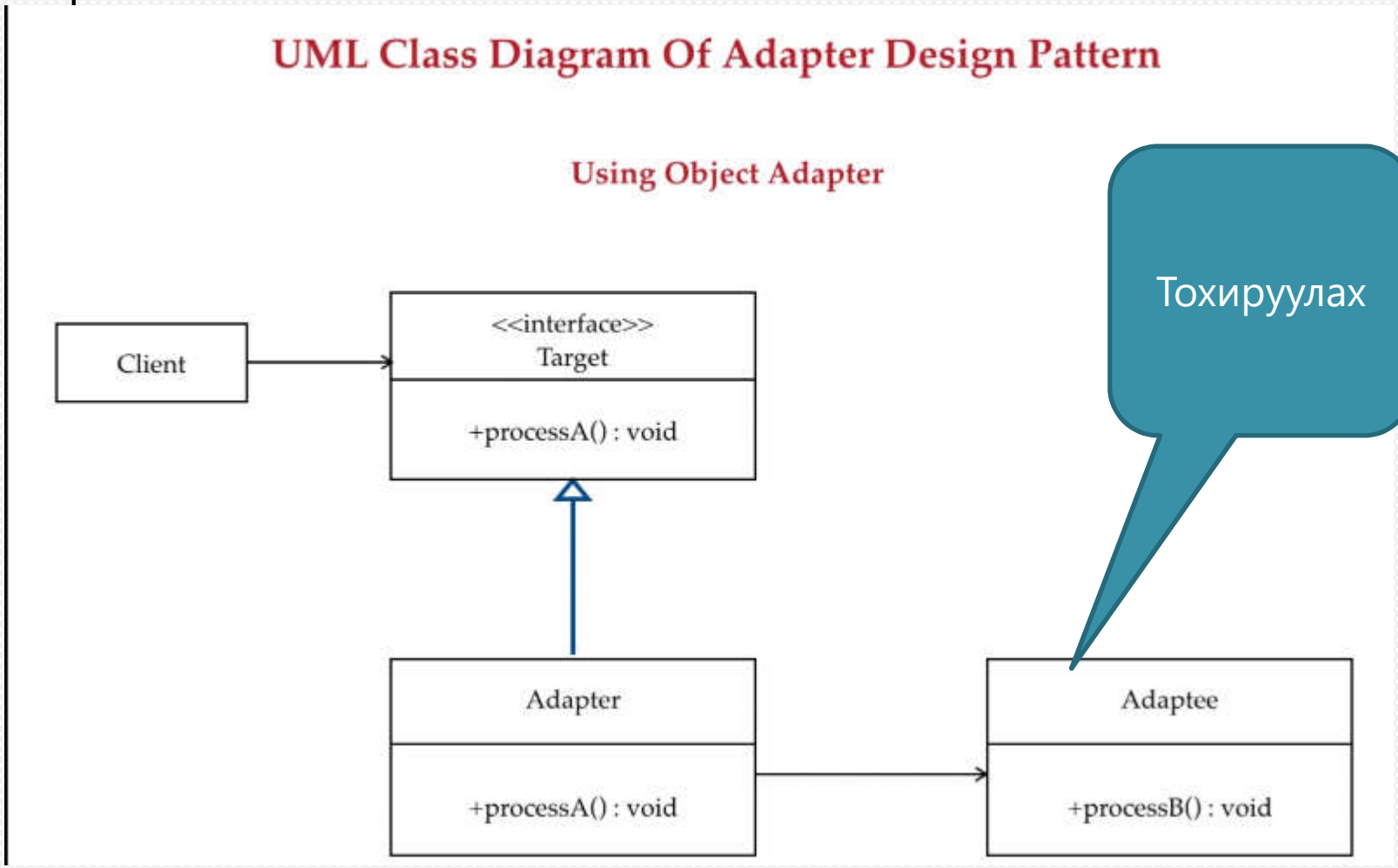
STRUCTURAL DESIGN PATTERN

- Хэрхэн үр ашигтай, хялбар, ахин ашиглагдах боломжтой байхаар обьект болон интерфэйсуудын хооронд харилцах харилцааны загварыг тодорхойлдог.

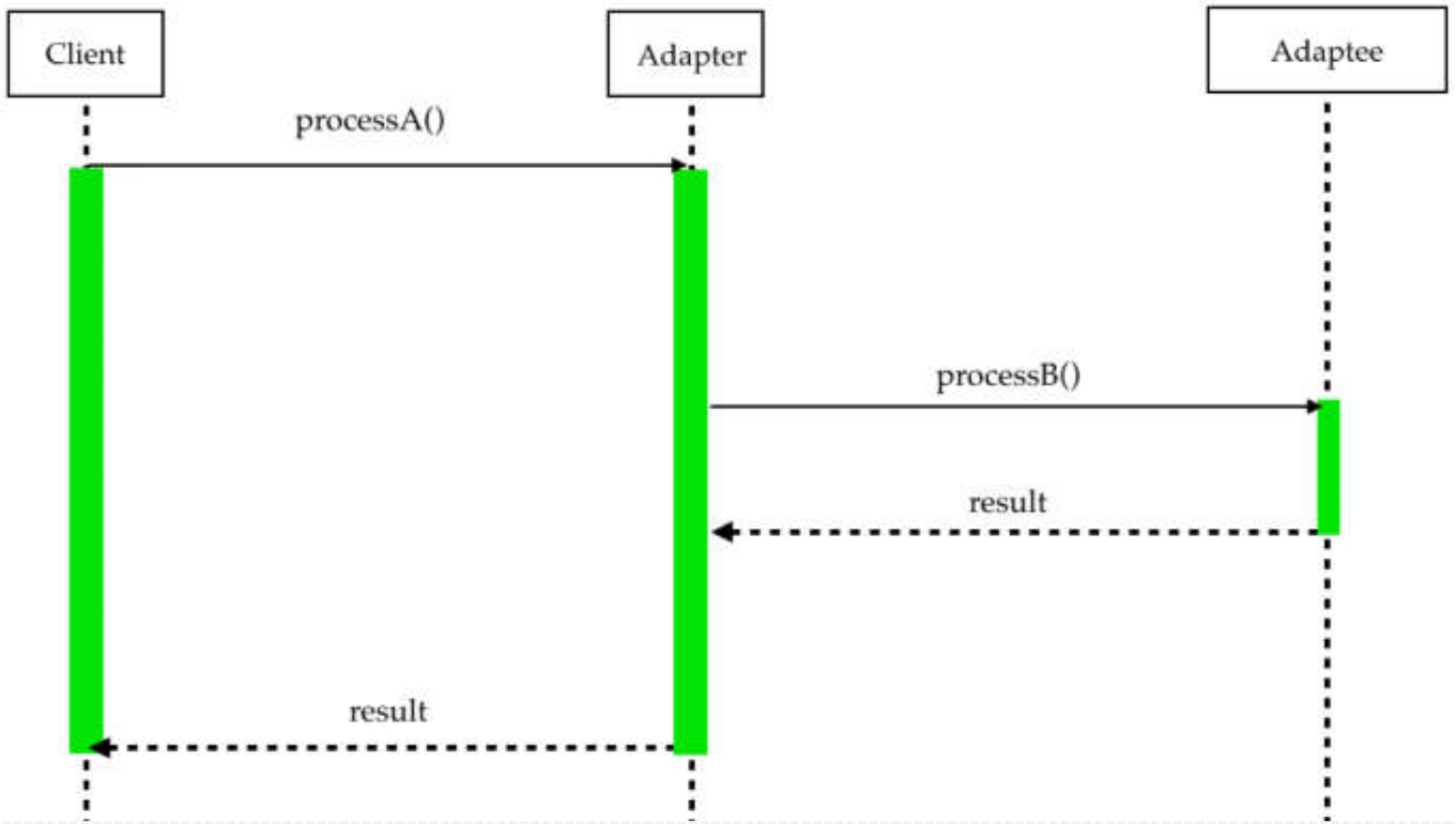
- Adapter Design Pattern
- Bridge Design Pattern
- composite Design pattern
- Decorator Design Pattern
- Facade Design Pattern
- Flyweight Design Pattern
- Proxy Design Pattern

ADAPTER DESIGN PATTERN

- Хамт ажиллах шаардлагатай 2 класс болон интерфэйсуудын кодыг өөрчлөхгүйгээр зохицуулалт хийх боломжийг олгодог design pattern юм.



UML Sequence Diagram Of Adapter Design Pattern



```
interface ISpeakFrench {
class FrenchtoEnglishAdapter
implements ISpeakEnglish{
ISpeakFrench french;
public
FrenchtoEnglishAdapter(ISpeakFrenc
h french) {
this.french=french;
}
@Override
public void speakEnglish() {
// TODO Auto-generated method st
this.french.speakFrench();
// Энэ хэсэгт бид францаар ярьса
зүйлийг англи руу хөрвүүлдэг код
бичиж өгөх хэрэгтэй.
System.out.println("Англиар
хэвлэгдэж байна.");
System.out.println("Францаар
хэлсэн зүйл англиар хөрвүүлэгдээ
```

```
interface ISpeakEnglish {
public void speakEnglish();


ass EnglishPerson implements
peakEnglish{

@Override
blic void speakEnglish() {
 TODO Auto-generated method stub
```

```
public class AdapterDemo {

public static void main(String[]
args) {
// TODO Auto-generated method stub
FrenchtoEnglishAdapter
translator=new
FrenchtoEnglishAdapter(new
FrenchPerson());
translator.speakEnglish();
```

```
<terminated> AdapterDemo [Java Application] C:\Program Files\Java\jre1.8.0_2
Францаар ярилаа.
Англиар хэвлэгдэж байна.
Францаар хэлсэн зүйл англиар хөрвүүлэгдээд хэвлэгдлээ.
```

When to use

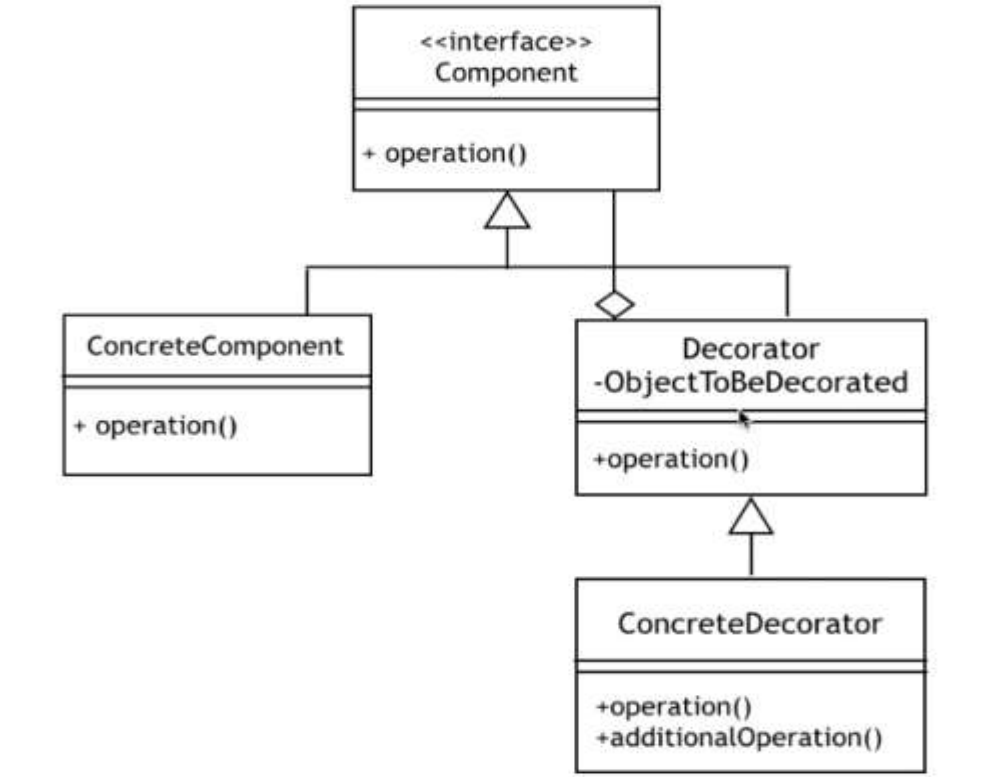
- There is an existing class, and its interface does not match the one you need.
- You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- There are several existing subclasses to be use, but it's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

Adapter Design Pattern in JDK

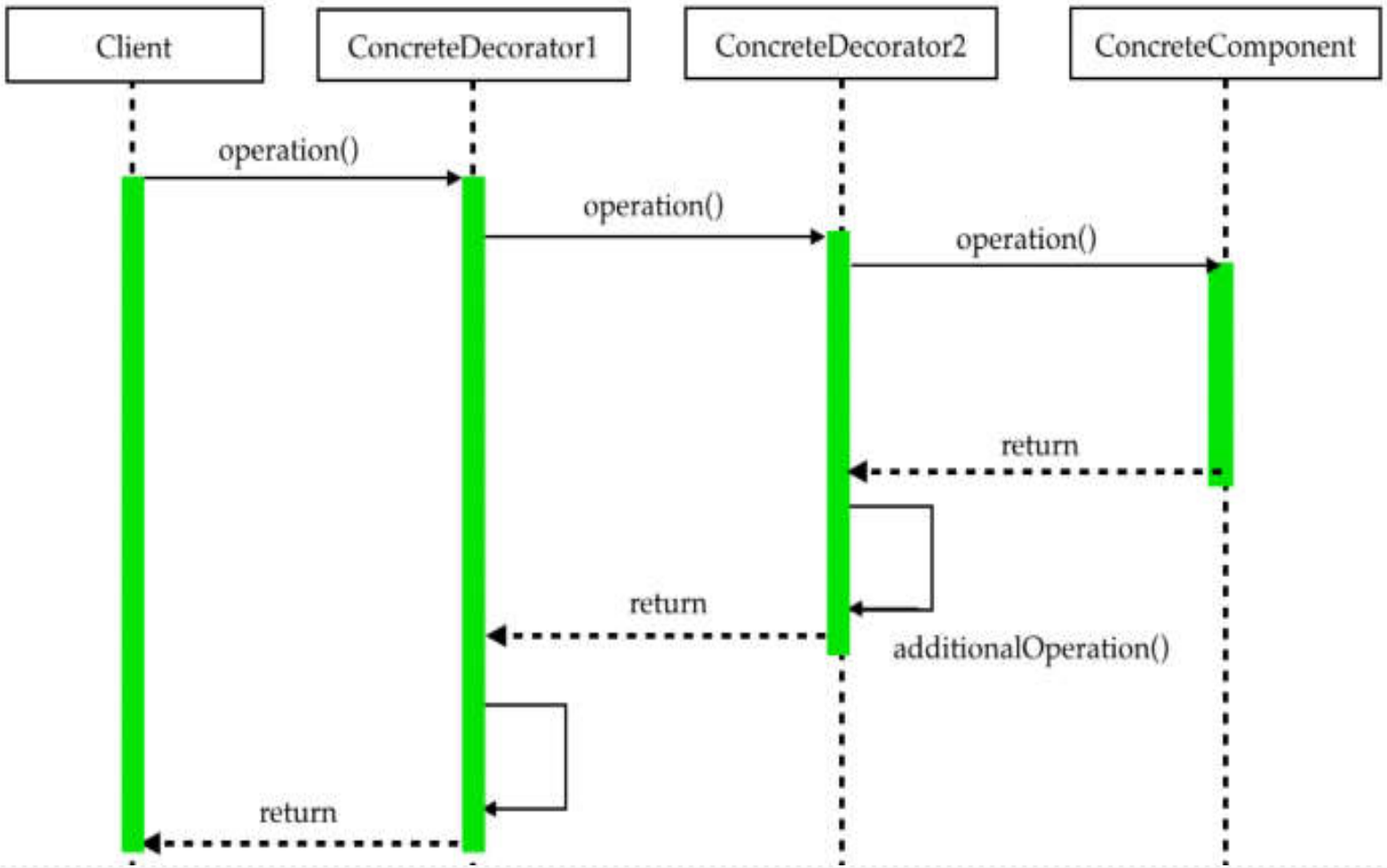
- java.util.Arrays#asList()
- java.util.Collections#list()
- java.util.Collections#enumeration()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)
- javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()

DECORATOR DESIGN PATTERN

- Өмнө нь тухайн обьектын бүтэц ямар байсаныг нь мэдэх шаардлагагүйгээр шинэ нэмэлт шинж чанаруудыг чимэглэх байдлаар нэмж өгөх боломжтой.



UML Sequence Diagram Of Decorator Design Pattern



```
interface Burger{
public void makeBurger();
}
```

```
class PlainBurger implements Burger{
public void makeBurger() {
System.out.println("Энгийн бургер
эн боллоо.");
```

```
abstract class BurgerDecorator
extends PlainBurger{
protected Burger bur

public BurgerDecorat
burger) {
```

```
public class
DecoratorDesignPattern {

public static void main(String[]
args) {
// TODO Auto-generated method stub
        Burger burger=new
PlainBurger();
        burger.makeBurger();
        new
CheeseBurgerDecorator(burger).make
Burger();
}

}
```

```
class CheeseBurgerDecor
extends BurgerDecorator
public
CheeseBurgerDecorator(B
burger) {
super(burger);
}
public void makeBurger(
burger.makeBurger();
System.out.println("Бяс
хийлээ.");
}
}
```

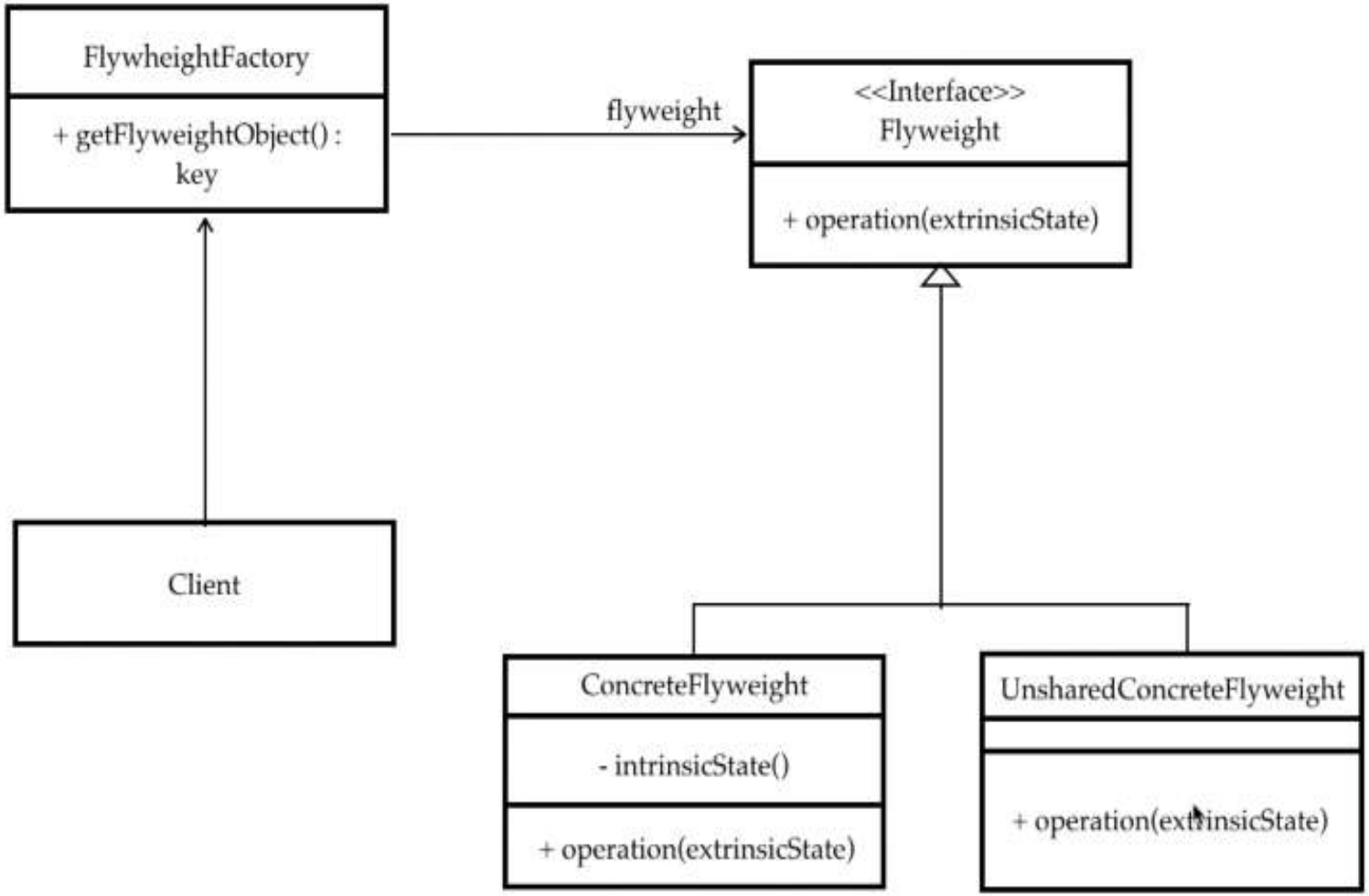
```
Decorator
tor{

tor(Burger

er() {

Чогоо болон
);
```

```
Энгийн бургер бэлэн боллоо.
Энгийн бургер бэлэн боллоо.
Бяслаг нэмж хийлээ.
```

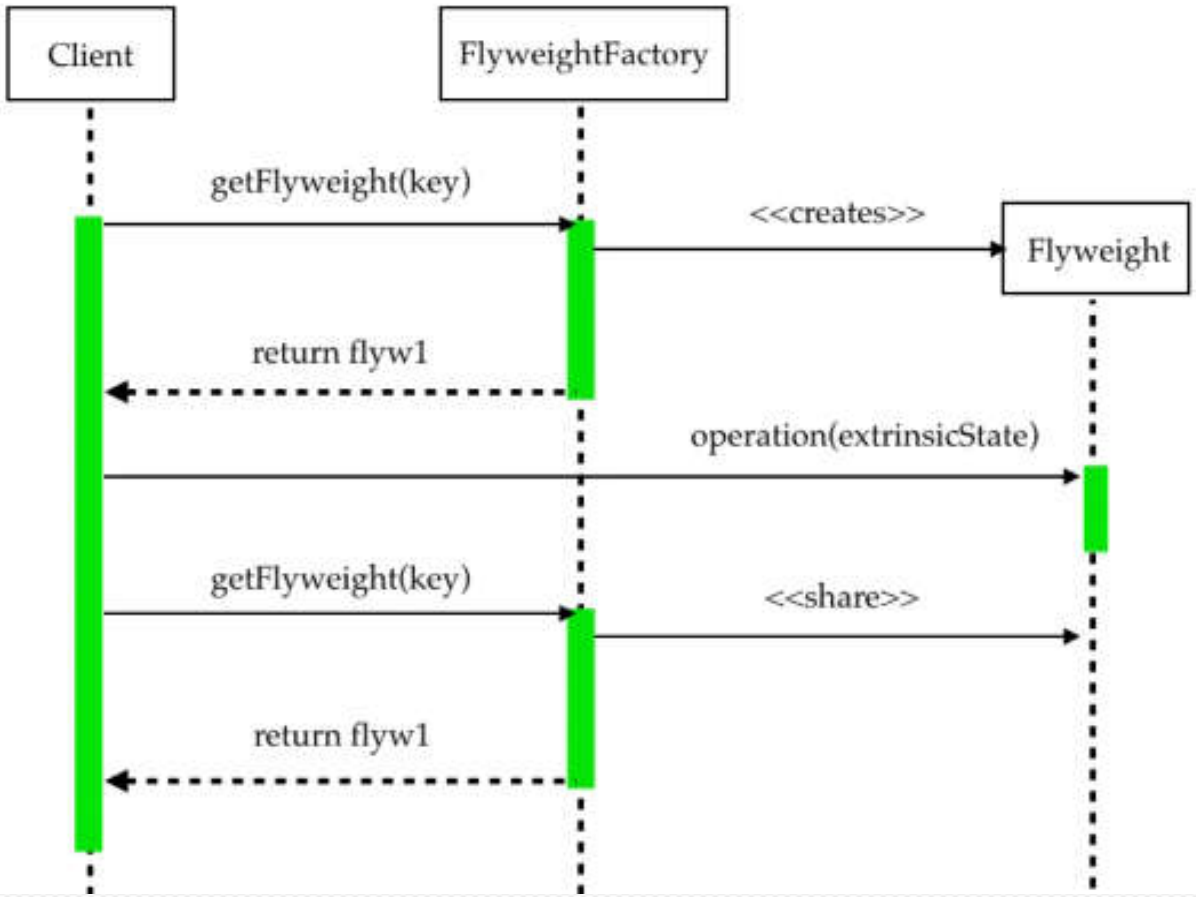
FLYWEIGHT DESIGN PATTERN

- Санах ойг хэмнэх зорилгоор нэг үүсгэсэн обьектоо хувааж ашиглах замаар ажилладаг.
- Их өгөгдөл бүхий олон обьектыг үүсгэх тохиолдолд хэдхэн ширхэгийг үүсгэчихээд ашиглана.

UML Class Diagram Of Flyweight Design Pattern



UML Sequence Diagram Of Flyweight Design Pattern



```
class Product{
private final String name;
public Product(String name) {
this.name=name;
}

public String toString() {
return name;
}
}
```

```
class Order{
private final int orderNumber;
private final Product product;
Order(int orderNumber, Product
product){
this.orderNumber=orderNumber;
this.product=product;
}

void processOrder() {
System.out.println(product +" энэ
бараанд захиалга хийгдлээ.
Захиалгын дугаар нь: " +
orderNumber);
}
}
```

```
class Portfolio{
private Map<String,Product>
products=new
HashMap<String,Product>();
public Product lookup(String
productName) {
if
(!products.containsKey(produ
)
products.put(productName, ne
Product(productName));
return products.get(productN
}

public int totalProductsMade
return products.size();
}
}
```

```
class Bucket{
private final Portfolio portfolio=new
Portfolio();
private final List<Order> orders=new
CopyOnWriteArrayList<Order>();
void executeOrder(String productName, in
orderNumber) {
Product
product=portfolio.lookup(productName);
Order order=new Order(orderNumber,
product);
orders.add(order);
}

void process() {
for (Order order:orders) {
order.processOrder();
orders.remove(order);
}
}

int getTotalProducts() {
return portfolio.totalProductsMade();
}
}
```

```java
public class FlyweightDesignPattern {

public static void main(String[] args) {
// TODO Auto-generated method stub
Bucket bucket=new Bucket();
bucket.executeOrder("Macbook pro",12);
bucket.executeOrder("Samsung-н гар утас",11);
bucket.executeOrder("Ухаалаг TV",7);
bucket.executeOrder("Угаалгын машин",5);
bucket.process();
System.out.println(bucket.getTotalProducts());
}

}
```

```
Macbook pro энэ бараанд захиалга хийгдлээ. Захиалгын дугаар нь: 12
Samsung-н гар утас энэ бараанд захиалга хийгдлээ. Захиалгын дугаар нь: 11
Ухаалаг TV энэ бараанд захиалга хийгдлээ. Захиалгын дугаар нь: 7
Угаалгын машин энэ бараанд захиалга хийгдлээ. Захиалгын дугаар нь: 5
4
```

When to use the Flyweight Design Pattern

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

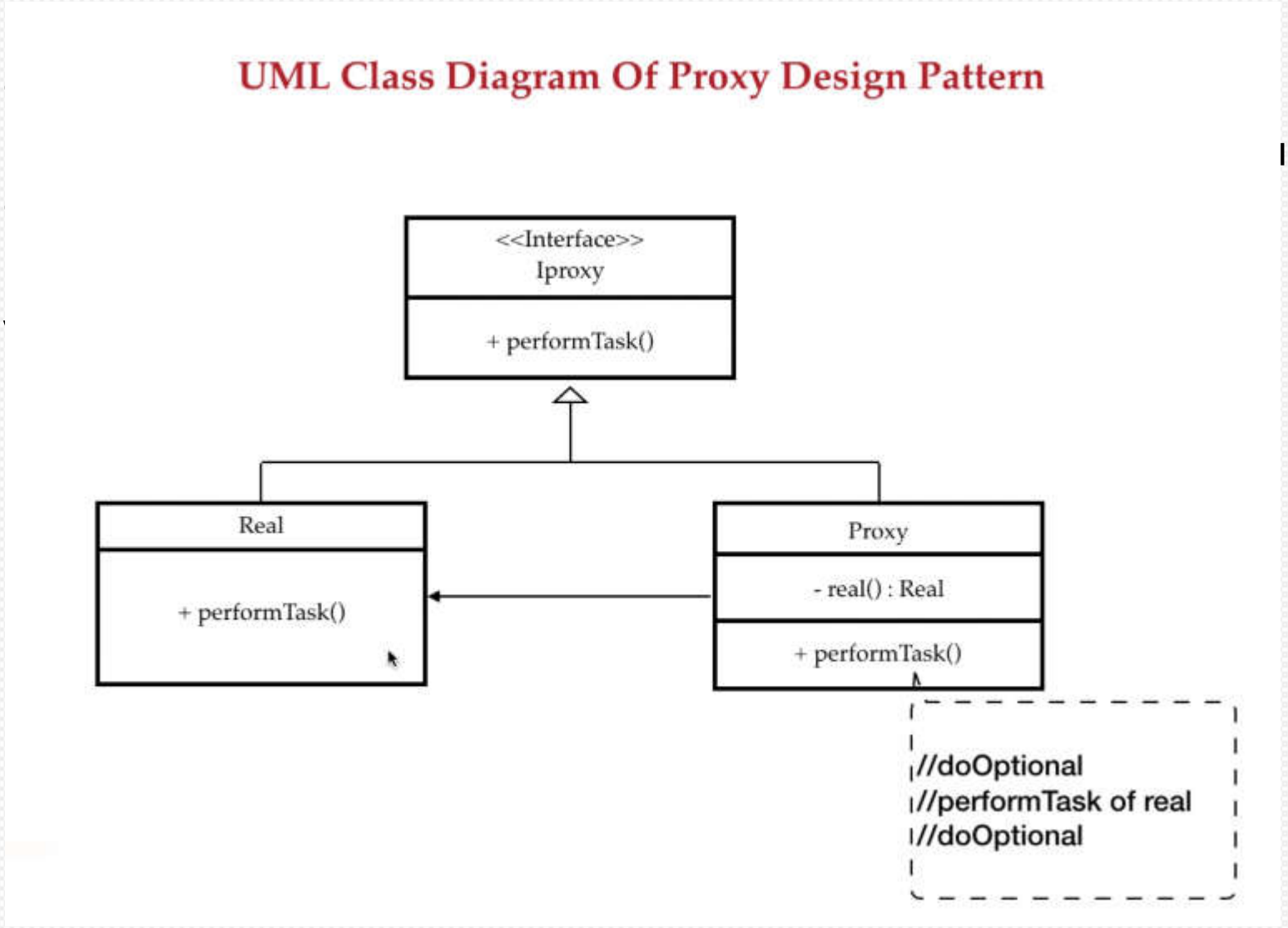
Flyweight Design Pattern in JDK

- java.lang.Integer#valueOf(int) (also on Boolean, Byte, Character, Short and Long)

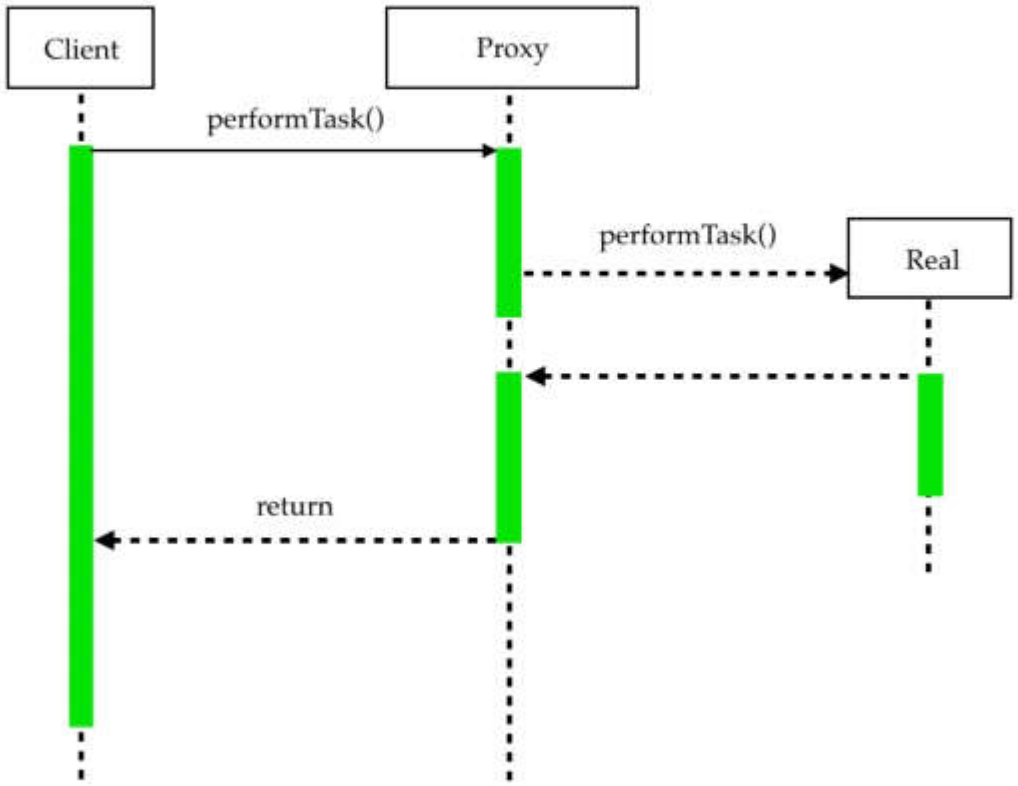
PROXY DESIGN PATTERN

-
-
-
-
-

IH



UML Sequence Diagram Of Proxy Design Pattern



```
interface Image{
public void displayImage();
}
```

```
class ActualImage implements Image{

@Override
public void displayImage() {
// TODO Auto-generated method stub
 бол оригналь зураг
```

```
public class ProxyDesignPattern {

public static void main(String[]
args) {
// TODO Auto-generated method stub
ProxyImage proxy = new
ProxyImage();
proxy.displayImage();

ProxyImageCache proxyCache=new
ProxyImageCache();
proxyCache.displayImage();
}
}
```

```
class ProxyImage
ActualImage real

@Override
public void disp
// TODO Auto-gen
System.out.print
эсэхийг шалгаж б
System.out.print
байна...");
realImage=new ActualImage();
realImage.displayImage();
}
}
```

```
eCache implements

lImage;

playImage() {
гдсэн эсэхийг
й бол доор байгаа
э.
tln("Обьект
йна. ");
tln("Бүх зүйл зөв
байна...");
realImage=new ActualImage();
realImage.displayImage();
}
}
```

When to use the Proxy Design Pattern

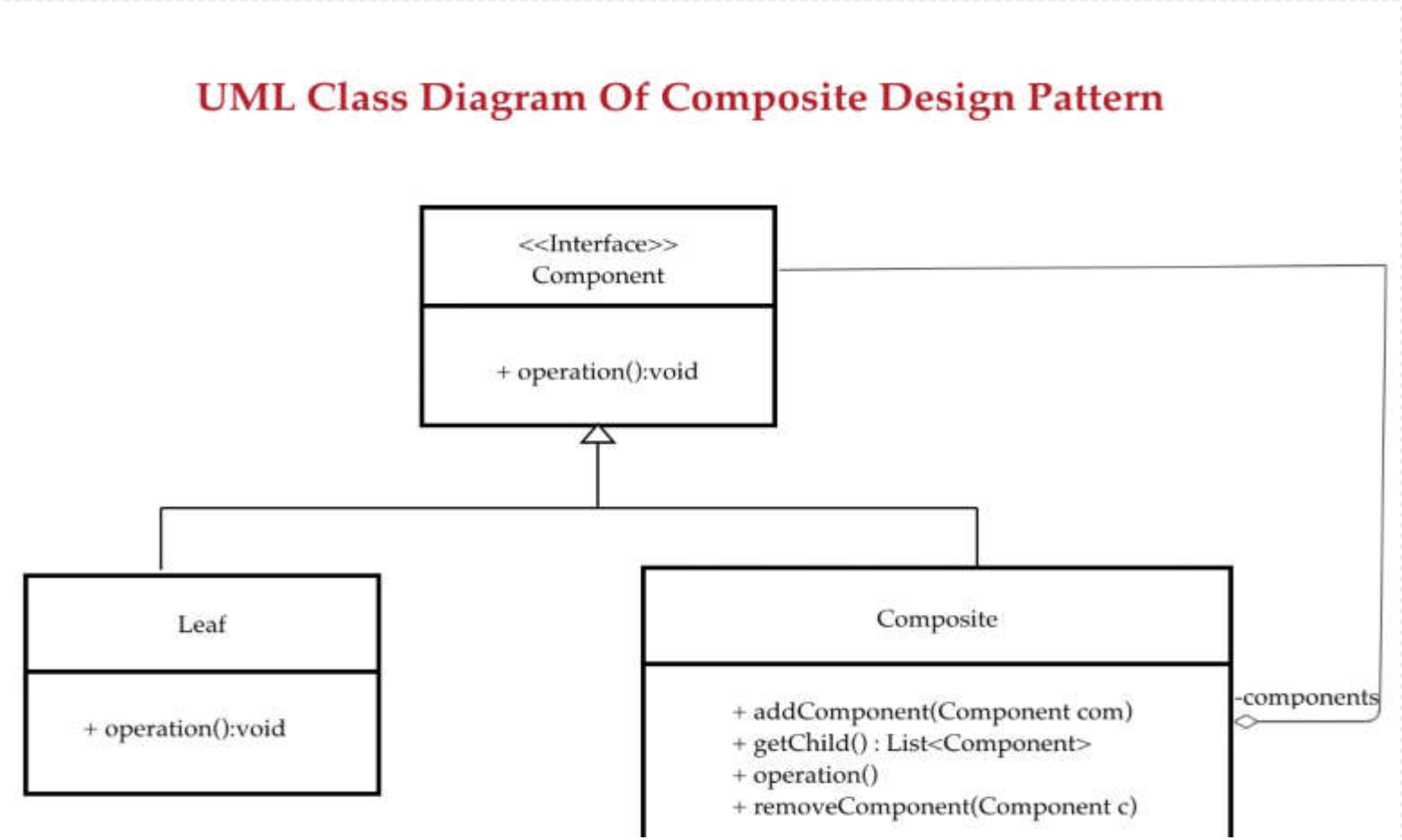
- A remote proxy provides a local representative for an object in a different address space.
- A virtual proxy creates expensive objects on demand.
- A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

Proxy Design Pattern in JDK

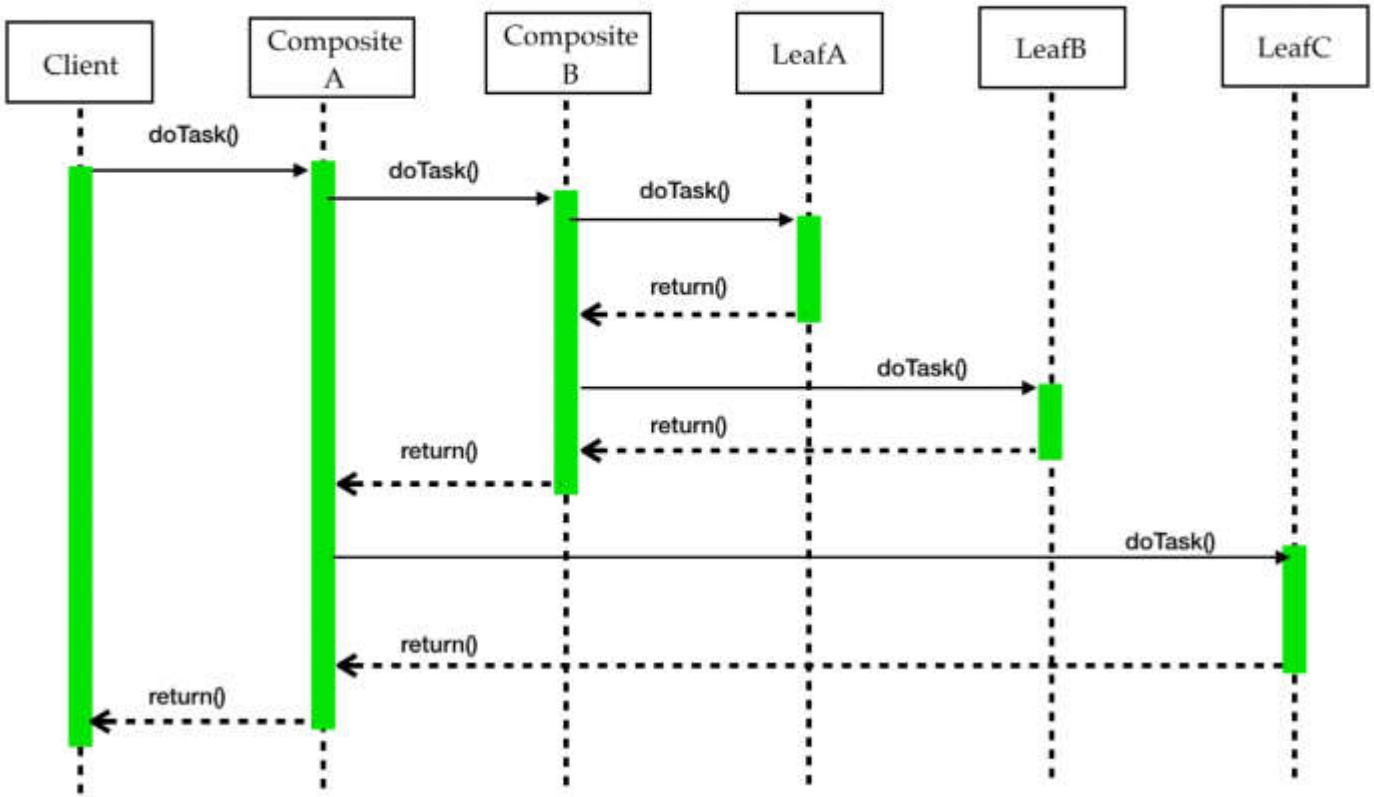
- java.lang.reflect.Proxy
- java.rmi.* (whole package)

COMPOSITE DESIGN PATTERN

- Нэг обьектыг үүсгэхдээ бүрэлдүүлж байгаа обьектуудын нийлмэл байдлаар үүсгэх боломжтой.
- Тухайн обьектыг үүсгэж байгаа дэд элементүүдийн мод бүтцийг харуулна.

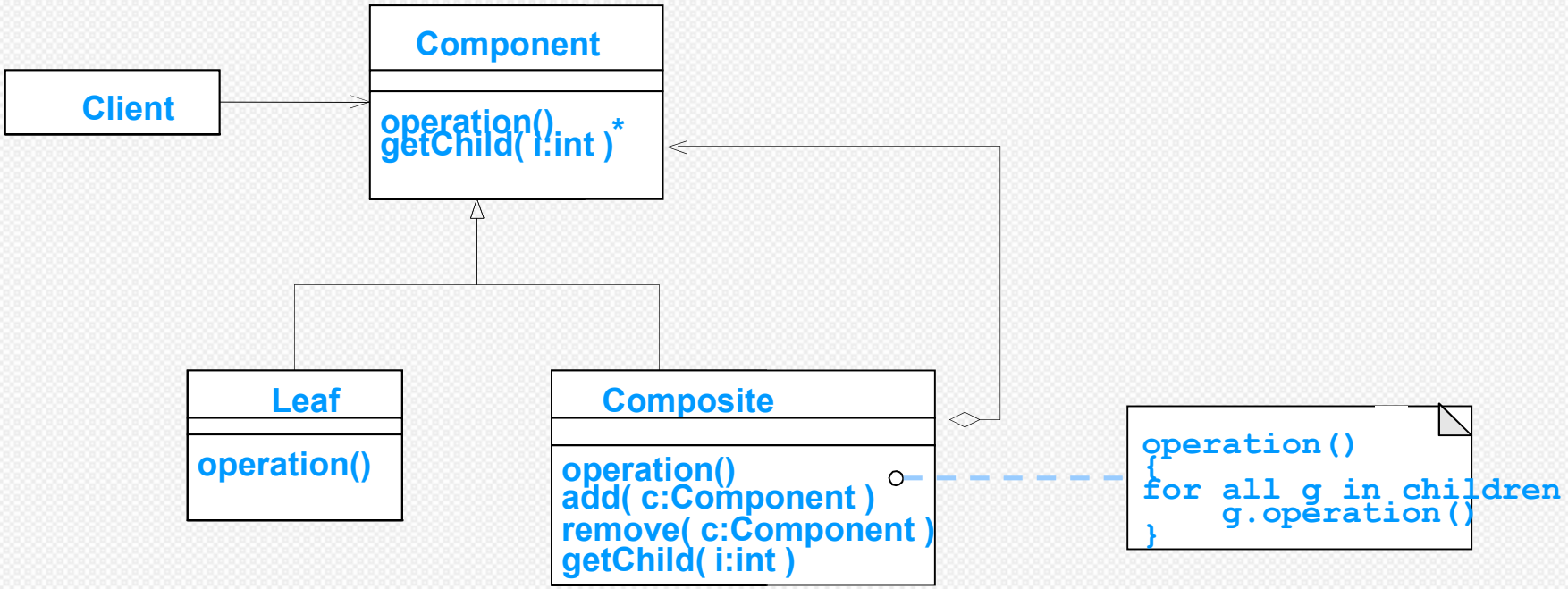


UML Sequence Diagram Of Composite Design Pattern



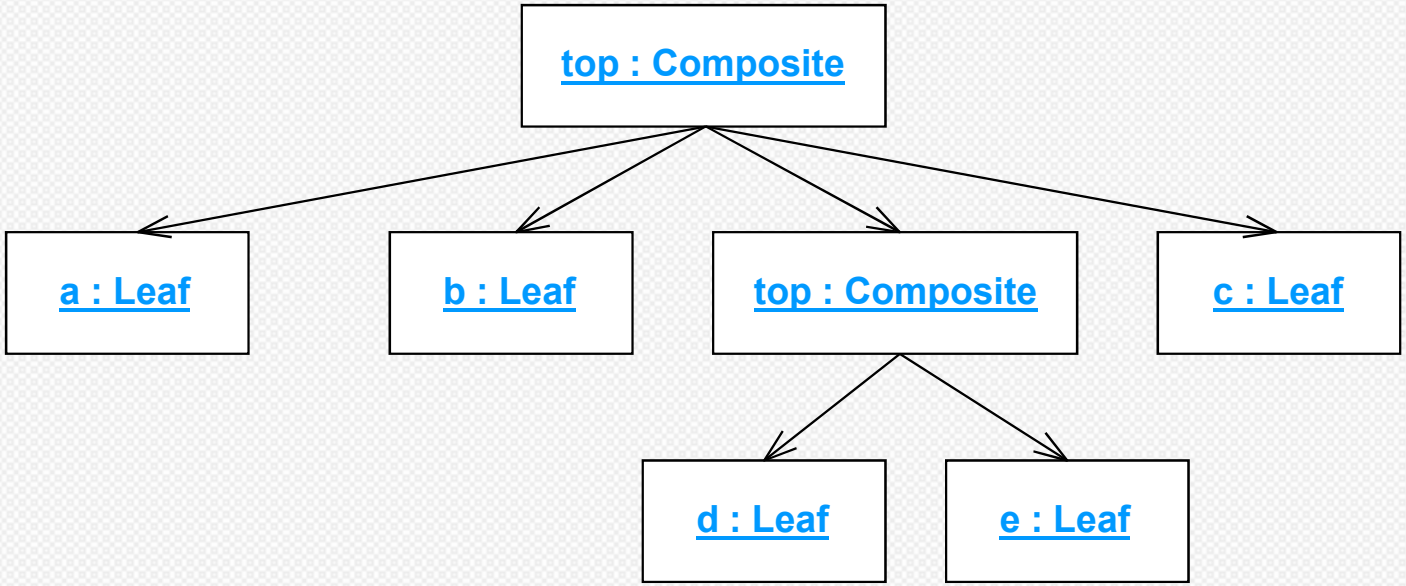
STRUCTURAL PATTERNS – COMPOSITE

Class Diagram



STRUCTURAL PATTERNS - COMPOSITE

Object Diagram



```
using System;
using System.Collections;

namespace DoFactory.GangOfFour.Composite.Structural
{

  // MainApp test application

  class MainApp
  {
    static void Main()
    {
      // Create a tree structure
      Composite root = new Composite("root");
      root.Add(new Leaf("Leaf A"));
      root.Add(new Leaf("Leaf B"));

      Composite comp = new Composite("Composite X");
      comp.Add(new Leaf("Leaf XA"));
      comp.Add(new Leaf("Leaf XB"));

      root.Add(comp);
      root.Add(new Leaf("Leaf C"));

      // Add and remove a leaf
      Leaf leaf = new Leaf("Leaf D");
      root.Add(leaf);
      root.Remove(leaf);

      // Recursively display tree
      root.Display(1);

      // Wait for user
      Console.Read();
    }
  }
```

```
// "Component"
  abstract class Component
  {protected string name;

    // Constructor
    public Component(string name)
    {this.name = name;}

    public abstract void Add(Component c);
    public abstract void Remove(Component c);
    public abstract void Display(int depth);
  }

  // "Composite"
  class Composite : Component
  {private ArrayList children = new ArrayList();

    // Constructor
    public Composite(string name) : base(name) {  }

    public override void Add(Component component)
    {children.Add(component);}

    public override void Remove(Component component)
    {children.Remove(component);}

    public override void Display(int depth)
    {Console.WriteLine(new String('-', depth) + name);

      // Recursively display child nodes
      foreach (Component component in children)
      {component.Display(depth + 2);}
    }
  }
```

```
-root
     ---Leaf A
     ---Leaf B
     ---Composite
     X
     -----Leaf XA
     -----Leaf XB
     ---Leaf C
```

```
// "Leaf"
  class Leaf : Component
  {// Constructor
    public Leaf(string name) : base(name) {  }

    public override void Add(Component c)
    {Console.WriteLine("Cannot add to a leaf");}

    public override void Remove(Component c)
    {Console.WriteLine("Cannot remove from a leaf");}

    public override void Display(int depth)
    {Console.WriteLine(new String('-', depth) + name);}
  }
}
```

Жишээ: Composite design pattern

```
interface Data{
public void doubleClick();
}
```

```
class Folder implements Data{
private String name;
    private List<Data> folder=new
ArrayList<Data>();
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
@Override
public void doubleClick() {
// TODO Auto-generated method stub
System.out.println(this.getName()+ "каталог
нээгдлээ.");
for (Data data : folder) {
data.doubleClick();
}
}
public void add(Data data) {
folder.add(data);
}

public void remove(Data data) {
folder.remove(data);
}}
```

```
class File implements Data{
private String name;

public String getName() {
return name;
}

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

@Override
public void doubleClick() {
// TODO Auto-generated method stub
System.out.println(this.getName()+
"файлыг хуганаар 2 дарж
нээлээ.");
}
}
```

```java
public class CompositeDesignPattern {

public static void main(String[] args) {
// TODO Auto-generated method stub
Folder f1=new Folder();
f1.setName("Folder 1 ");
Folder f2=new Folder();
f2.setName("Folder 2 ");

File file1=new File();
file1.setName("Файл 1 ");
File file2=new File();
file2.setName("Файл 2 ");
File file3=new File();
file3.setName("Файл 3 ");
File file4=new File();
file4.setName("Файл 4 ");

f1.add(file1);
f2.add(file2);
f1.add(file3);
f2.add(file4);

f1.doubleClick();
f2.doubleClick();
}

}
```

```
Folder 1 каталог нээгдлээ.
Файл 1 файлыг худганаар 2 дарж нээлээ.
Файл 3 файлыг худганаар 2 дарж нээлээ.
Folder 2 каталог нээгдлээ.
Файл 2 файлыг худганаар 2 дарж нээлээ.
Файл 4 файлыг худганаар 2 дарж нээлээ.
```

When to use the Composite Design Pattern

- When you want to represent part-whole hierarchies of objects.
- When you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite Design Pattern in JDK

- java.awt.Container#add(Component)
- javax.faces.component.UIComponent#getChildren()

Анхаарал хандуулсанд баярлалаа