STRUCTURAL ЗОХИОМЖИЙН ПАТТЕРН

Structural буюу бүтцийн паттерн нь:

- Объект хоорондын хариуцлага хүлээлгэх үйл ажиллагаанд хамааралтай. Үр дүн нь хоорондоо углуурга багатай бүрэлдэхүүн хэсэг бүхий давхарласан архитектур болно.
- Нэг объектод нөгөөгөөс энгийн аргаар хандаж болохгүй үед эсвэл объектын интерфейс таарахгүйн улмаас ашиглаж болохгүй үед объект хоорондын харилцааг хангаж өгөх.

Ерөнхийдөө нэгж хоорондын харилцааг зохицуулснаар зохиомжийг хөнгөвчлөх зорилготой юм.

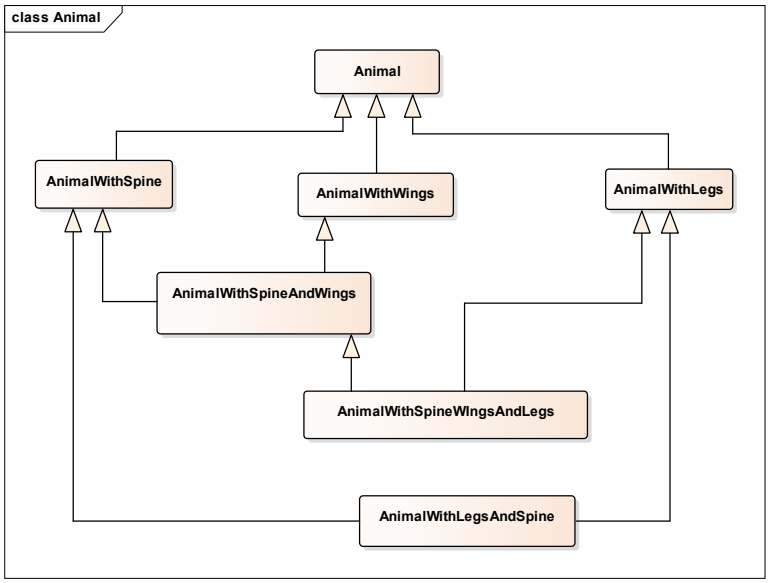
DECORATOR

Чимэгч буюу decorator паттернийг эх классыг нь өөрчлөхгүй, удамшил ашиглахгүйгээр объектын үйл ажиллагааг динамикаар ихэсгэхэд ашигладаг. Үүнийг decorator гэх объект ороогч үүсгэж гүйцэтгэдэг.

- Decorator объект нь доор орших объекттойгоо ижил интерфейстэй байна гэж зохиомжилдог. Энэ нь клиент объектод decorator болон доор орших бодит объекттой яг адил арга замаар хандах боломж олгодог.
- Decorator объект нь бодит объектын reference агуулахгүй.
- Decorator объект клиентээс бүх хандалт(дуудлага) хүлээж авдаг. Эдгээр дуудлагыг дараа нь доор орших объектод явуулдаг.

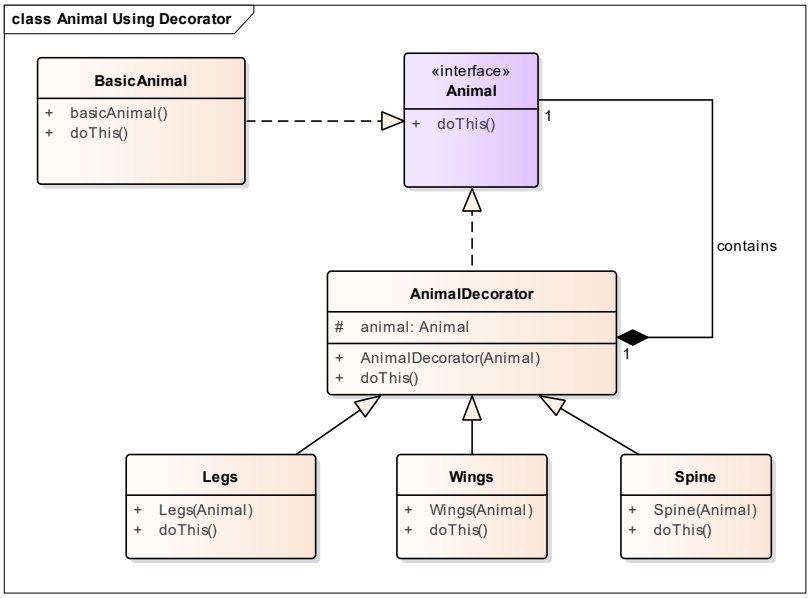
Ихэвчлэн объект хандлагат зохиомжийн хувьд классыг өөрчлөхгүйгээр шинэ ажиллагаа нэмэхийн тулд удамшил ашиглах шаардлагатай байна.

Animal класс нь spine, wings эсвэл legs гэсэн дэд классуудтай. Тухайн дэд классуудаас дурын нэг класс аль нэг эсвэл бүгдээс нь удамшсан байж болно. Энэ тохиолдолд дараах класс диаграм гарч ирнэ. Энэ аргаар классуудыг зохион байгуулбал дэд класс нэмэх бүрт классын шатлал маш ихээр тэлэх шаардлагатай болно.



Диаграм 1

Дээрх класс диаграмын асуудлыг decorator паттерн ашиглан шийдвэрлэх юм бол:



Диаграм 2

- AnimalDecorator нь Animal биеллийг заасан reference агуулна. Энэ reference нь ороож буй Animal объектыг заана.
- AnimalDecorator нь Animal интерфейсийг хэрэгжүүлдэг бөгөөд ороож буй объектдоо ирж буй дуудлагыг дамжуулдаг doThis() методын үндсэн хэрэгжүүлэлтийг өөртөө агуулна. Тиймээс AnimalDecorator-ын бүх дэд класс нь doThis методыг дотроо тодорхойлсон байна гэсэн үг.
- Decorator нь объектод динамикаар нэмэлт үүрэг олгодог. Animal нь Legs, Wings, Spine гэх мэт decorator хэдийг ч нэмсэн Animal хэвээрээ байна.

Дээрхээс харвал decorator зохиомжийн паттерныг дараах төрлүүдээс хэрэгжүүлнэ.

1. Бүрэлдэхүүн хэсгийн интерфейс – Бидний методын хэрэгжүүлэлтийг тодорхойлох интерфейс эсвэл хийсвэр класс болно.

```
public interface Animal {

	public void doThis();
}
```

2. Бүрэлдэхүүн хэсгийн хэрэгжүүлэлт – интерфейсийн үндсэн хэрэгжүүлэлт.

```
public class BasicAnimal implements Animal{
		@Override
		public void doThis() {
			System.out.print("Basic Animal. ");
		}
}
```

3. Decorator – decorator класс нь интерфейсийг хэрэгжүүлэх бөгөөд бүрэлдэхүүн хэсгийн интерфейстэй байдаг гэсэн харилцаатай байна.

```
public class AnimalDecorator implements Animal{
	protected Animal animal;

	public AnimalDecorator(Animal a){
		this.animal=a;
	}

	@Override
	public void doThis() {
		this.animal.doThis();
	}
}
```

4. Бодит decorator – Суурь decoratorийн ажиллагааг сунгаж, бүрэлдэхүүн хэсгийн шинжийг зөв зохистой өөрчилнө. Legs, Wings, Spine классууд юм.

```
public class Legs extends AnimalDecorator{

	public Legs(Animal a) {
		super(a);
	}

	@Override
	public void doThis(){
		super.doThis();
		System.out.print("This animal has Legs. ");
	}
}
```

Дээрх програмын ажиллагааг харж үзье:

```
Animal animal = new BasicAnimal();
animal.doThis();
```

Үндсэн хэрэгжүүлэлт дээр зааж өгсөн методыг дуудсан учраас “*Basic Animal*.” гэсэн үр дүн хэвлэнэ.

```
Animal spineAnimal = new Spine(new BasicAnimal());
spineAnimal.doThis();
```

BasicAnimal дээр Spine гэсэн шинжийг нэмж өгч байна. Энэ тохиолдолд "*Basic Animal. This animal has Spine.*" гэж хэвлэнэ.

```
Animal wingsSpineAnimal = new Wings(spineAnimal);
wingsSpineAnimal.doThis();
```

Өмнө үүсгэсэн объект дээрээс шинэ объект үүсгэж байна. Энэ тохиолдолд "*Basic Animal. This animal has Spine. This animal has Wings.*" гэж хэвлэнэ.

```
Animal allAnimal = new Spine(new Wings(new Legs(new BasicAnimal())));
allAnimal.doThis();
```

Дээрх тохиолдолд "*Basic Animal. This animal has Legs. This animal has Wings. This animal has Spine.*" гэж хэвлэнэ. Програмын runtime үед төрөл бүрийн объект болон дараалал үүсгэж болохыг анзаарч болно.

ADAPTER

Adapter паттерн нь тухайн хэрэглээнд таарахгүй объектын интерфейсийг ороосон adapter паттерн тодорхойлдог. Ороож буй объектыг adapter гээд ороогдож буйг нь adaptee гэнэ. Adapter нь хоорондоо тохирохгүй интерфейстэй хоёр классыг хамтран ажиллуулах боломж олгоно.

Дээрх тодорхойлолтод ашигласан интерфейс бол:

- Java програмчлалын хэлний интерфейс биш.
- GUI-ийн интерфейс биш.
- Классын ил гаргах програмчлалын интерфейс буюу бусад класс ашиглахад зориулагдсан интерфейсийг хэлнэ. Жишээ нь, класс нь хийсвэр класс эсвэл, интерфейс гэж зохиомжлоход зарласан методын олонлогийг классын интерфейс гэнэ.

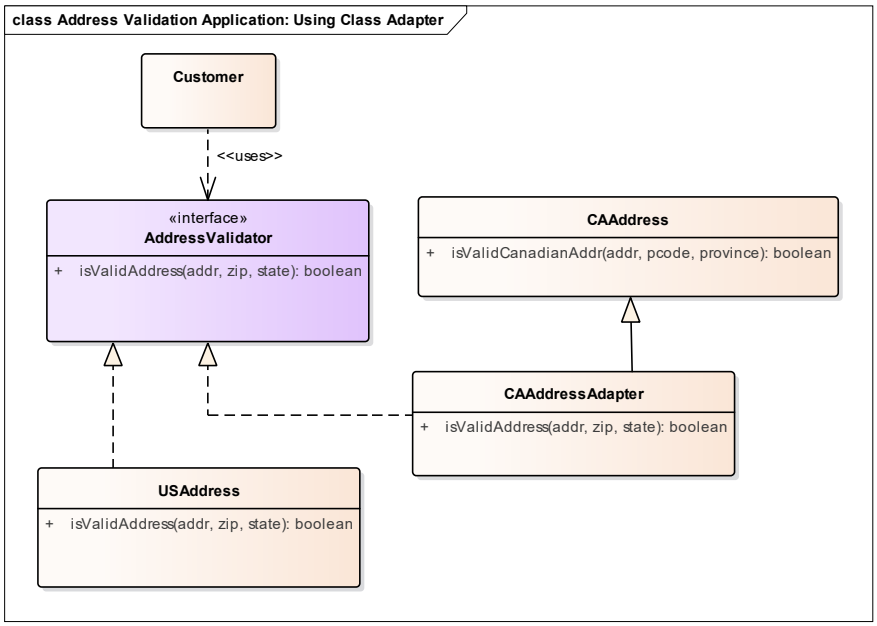
Adapter нь ерөнхийдөө класс болон объект adapter гэсэн хоёр ангилалд хуваагддаг.

Класс adapter

Энэхүү Adapter нь adaptee классуудыг дэд класс болгож зохиомжилдог. Нэмэхэд, класс adapter нь клиент объектод нийцсэн интерфейсийг хэрэгжүүлдэг. Клиент объект нь класс adapter методыг дуудахад adapter дотоодоор уламжилж авсан adaptee методыг дуудна.

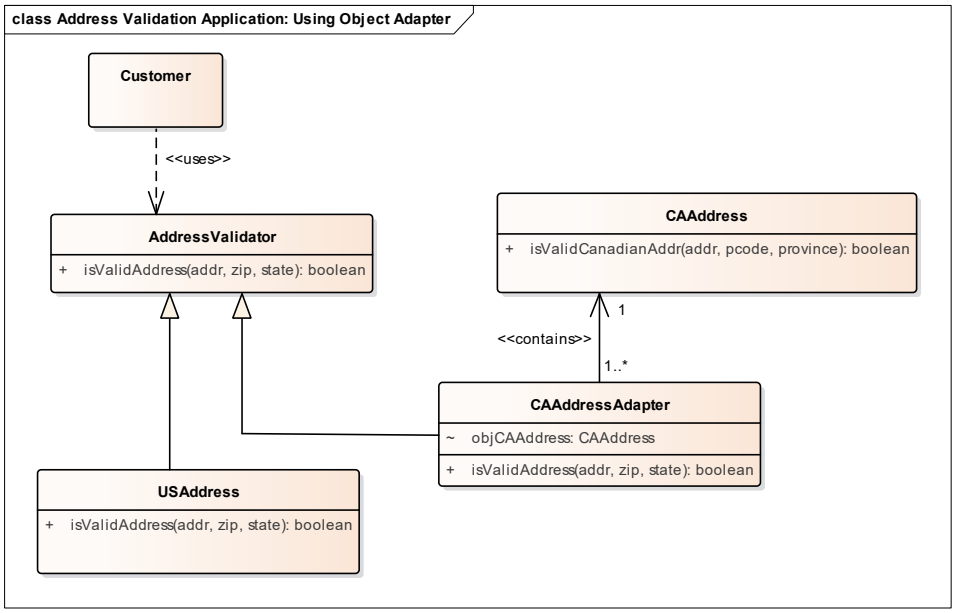
Объект adapter

Объект adapter нь ороож буй классынхаа биеллийг агуулна. Энэ тохиолдолд, adapter нь ороогдож буй классынхаа биелэлд дуудлага хийнэ.



Диаграм 3

Диаграм 3-т Address Validation application-ын класс диаграмыг дүрсэлсэн байна. CAAdress нь isValidCanadianAdd(addr, pcode, province) методтой байна. Гэхдээ энэ метод нь AddressVali-dator интерфейст таарахгүй учраас CAAddressAdapter классаар ороож байна. Ингэж хийсэн тохиолдолд класс adapter ашигласан гэж үзнэ.



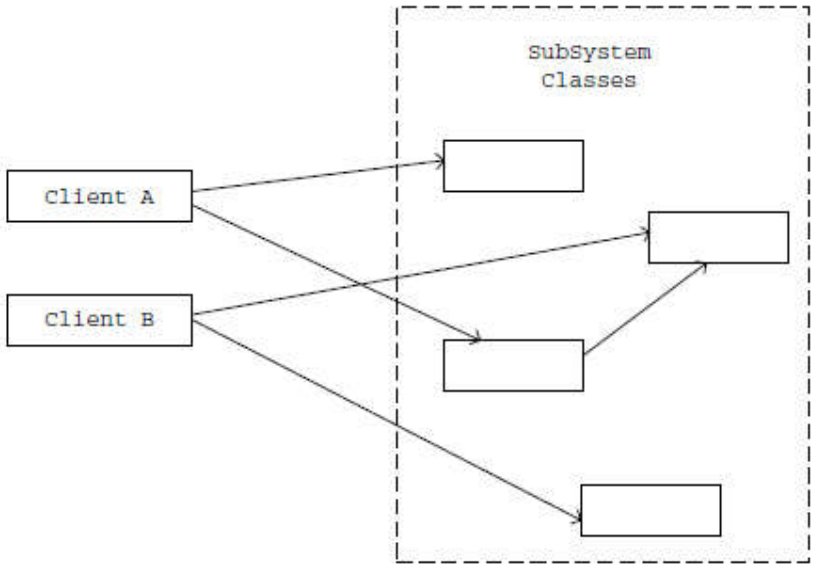
Диаграм 4

Дээрх Диаграмт өмнөх application-ийг объект adapter ашиглан зохиомжилсон байгааг харж болно. Клиент объект CAAddressAdapter дээр isValidAddress методыг дуудахад, adapter дотооддоо CAAddress биелэл дээрх isValidCanadianAddr методыг дуудна. Энэ нь Adapter нь ороож буй объектын биеллийг хадгалдаг учраас боломжтой юм.

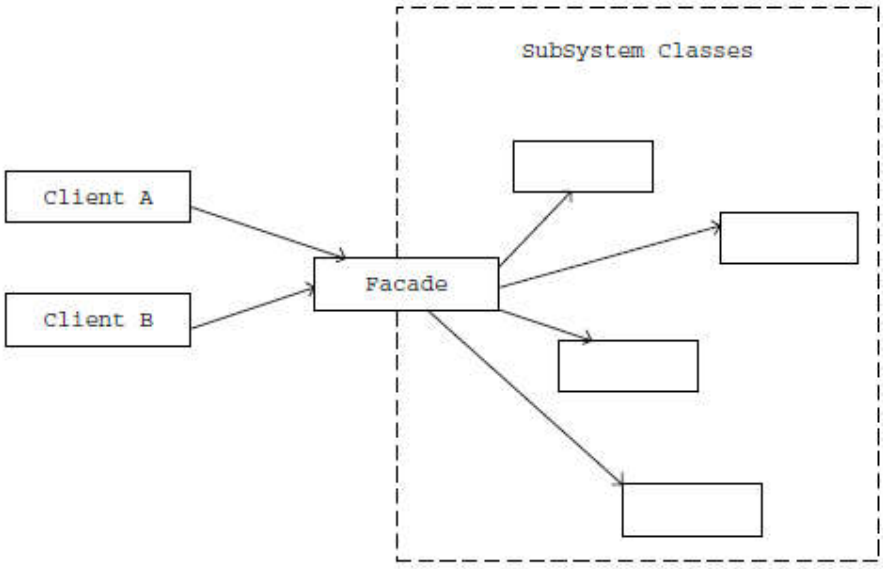
FAÇADE

Дэд системийн интерфейсийн олонлогийг нэгдсэн интерфейсээр хангана. Façade нь дэд системийг ашиглахад амар болгох дээд-төвшин бүхий интерфейсийг тодорхойлно.

Системийг дэд систем болгон хувиргаж бүтэц үүсгэснээр төвөгтэй байдлыг нь багасгадаг. Зохиомжийн нийтлэг зорилт бол дэд систем хоорондын харилцаа болон хараат байдлыг багасгах юм. Энэ зорилтыг биелүүлэх нэг арга гэвэл façade гэх дэд системийг хөнгөвчилсөн интерфейс бүхий объект танилцуулах юм.

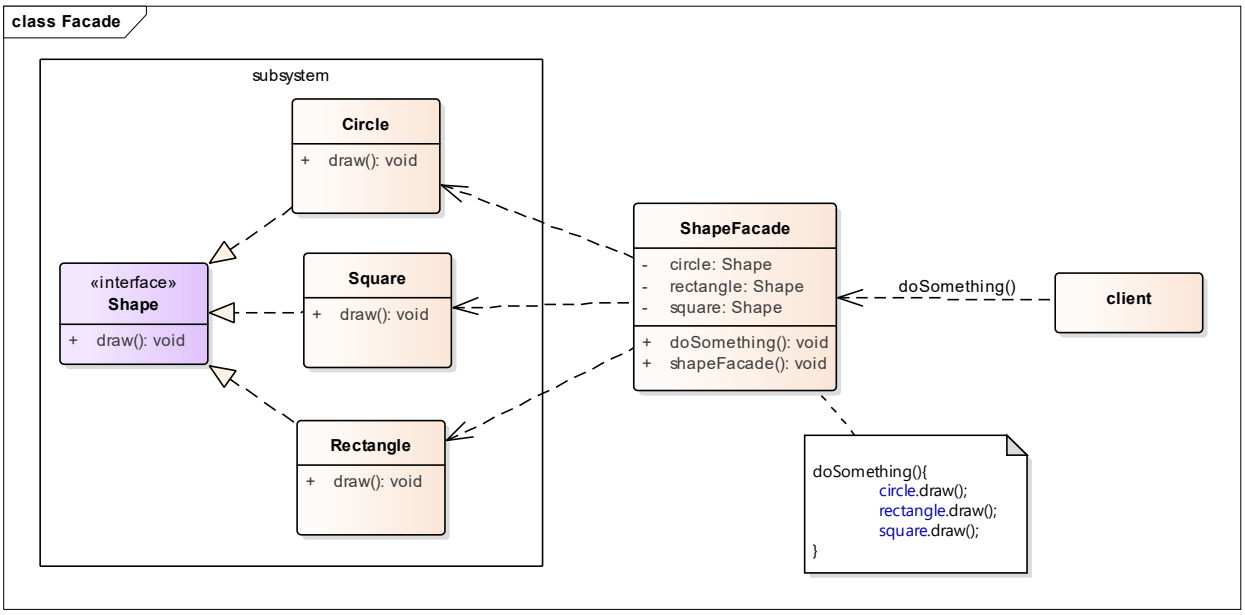


Диаграм 5



Диаграм 6

Дээрх диаграмууд дээр Façade-ийн ажиллагааг дүрсэлсэн болно.



Диаграм 7

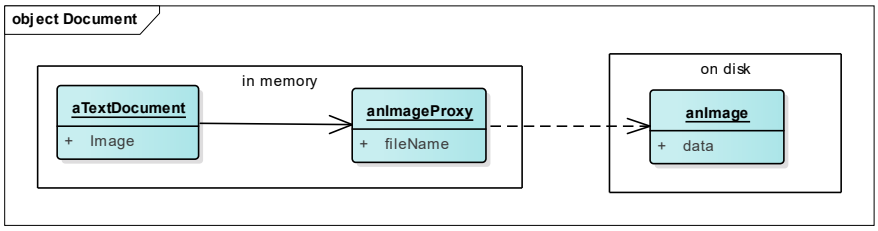
ShapeFacade жишээг дээрх класс диаграмт дүрслэв. Square, Circle, Rectangle нь бол shape subsystem-д хамаардаг. ShapeFacade класс нь клиент бүрт shape дэд системийн энгийн нэг л интерфейс олгож байна. Ингэснээр клиентийг дэд системийн бүрэлдэхүүн хэсгүүдээс халхалж харилцан ажиллах объектыг багасгаснаар дэд системийг ашиглахад амар болгох юм.

PROXY

Proxy нь объектод орлуулагч эсвэл түр зуурийн байр эзлэгч тавьж өгч хандалтыг хянахад хэрэг болдог. Объектын хандалтыг хязгаарлах нэг шалтгаан бол үүсгэх болон бэлтгэх үеийн нийт өртөгийг ашиглах хүртлээ багасгах юм. Жишээ нь бичиг баримт боловсруулдаг програм авч үзье. Зарим зураг, жишээлбэл растер зураг боловсруулах өртөг өндөр. Гэхдээ бичиг баримт хурдан нээгдэж ажиллахад бэлэн болох ёстой, тиймээс бид баримтыг нээх үед бүх зургийг боловсруулахаас татгалзах хэрэгтэй. Угаасаа нэг дор эдгээр объект бүгд дүрслэгдэхгүй.

Эдгээр хязгаарлалт нь өртөг өндөр объектыг шаардахад нь буюу зураг харагдах тохиолдолд үүсгэхийг санал болгодог. Гэхдээ зургийн оронд баримтад юу тавих вэ? Бас редакторын хэрэгжүүлэлтийг будлиантай болгохгүйн тулд зургийг шаардахад л үүсгэж байгаа гэдгийг хэрхэн нуух вэ? Мөн энэ оновчлол нь үзүүлэх болон форматад оруулах кодод нөлөөлөхгүй байх ёстой.

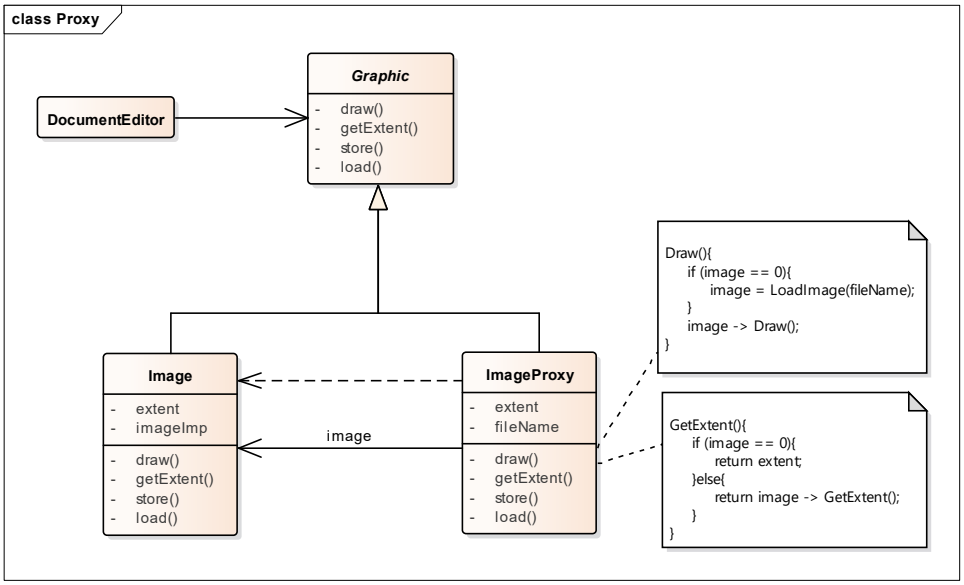
Шийдэл нь гэвэл зургийн proxy гэх жинхэнэ зургийн оронд түр байрлах объект юм. Энэ proxy нь тухайн зураг шиг л төлөвтэй бөгөөд шаардлагатай үед нь үүсгэж чаддаг байна.



Диаграм 8

Image proxy нь бичиг баримт боловсруулагч өөрийгөө үзүүл гэж Draw методыг нь дуудахад жинхэнэ зургийг үүсгэнэ. Proxy нь дараа дараагийн хүсэлтийг зурагт шууд дамжуулна. Тиймээс зургийн хаяг бүхий таних тэмдгийг proxy нь хадгалах хэрэгтэй.

Зураг нь өөр өөр файлд хадгалагдсан гэж үзье. Энэ тохиолдолд зургийн нэрийг жинхэнэ объектийн таних тэмдэг болгож ашиглаж болно. Proxy нь extent-ийг нь хадгална, жишээ нь өндөр болон урт. Extent нь proxy-оос зургийн хэмжээг асуухад зургийг үүсгэлгүйгээр хариу өгөхөд хэрэг болдог.



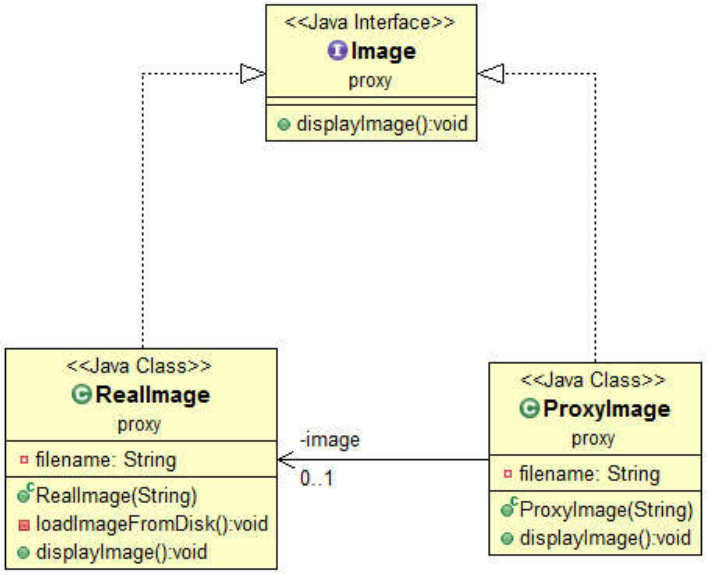
Диаграм 9

Document editor нь оруулж өгсөн зургийг хийсвэр класс бүхий Graphic интерфейсээр хандана. ImageProxy нь шаардлагаар үүссэн зургийн класс юм. ImageProxy нь диск дээр байрлах зургийн таних тэмдэг болох filename-ийг агуулна. Файлийн нэр нь ImageProxy-ийн байгуулагчид аргумент болон дамжуулагдана.

ImageProxy нь мөн зургийн хэмжээ болон Image биеллийн референцийг агуулна. Энэ биелэл нь proxy зургийг жишээгээр үүсгэх хүртэл хүчингүй байна. Draw метод нь хүсэлтийг илгээхээс өмнө зургийг жишээгээр үүсгэсэн байгааг шалгана. GetExtent нь зөвхөн зураг жишээгээр үүссэн үед хүсэлтийг дамжуулна. Хэрвээ үүсээгүй бол ImageProxy-д хадгалагдах extent-ийг буцаана.

Proxy-г ашиглах хэд хэдэн тохиолдол байна. Эдгээр нь:

- **Remote Proxy** нь өөр хаягийн зайд орших объектын дотоод төлөөллийг олгоно. Жишээ нь ATM нь хол сервер дээр орших банкны мэдээллийг хадгалдаг proxy бүхий объекттой байж болно.
- **Virtual Proxy** өртөг өндөр объектыг хэрэгцээтэй үед үүсгэдэг. Өмнө тайлбарласан ImageProxy нь үүнд хамаарна. Өөр нэгэн ашиглалтын жишээ гэвэл:



Диаграм 10

Дээрх жишээ нь объектын displayImage гэсэн методыг ажиллуулахад нэг удаа объектыг үүсгээд цаашид үүсгэсэн объектоо харуулдаг байна.

```
Image image1 = new ProxyImage("HiRes_10MB_Photo1");
final Image image2 = new ProxyImage("HiRes_10MB_Photo2");

image1.displayImage(); // үүсгэх хэрэгтэй
image1.displayImage(); // үүсгэх хэрэггүй
image2.displayImage(); // үүсгэх хэрэгтэй
image2.displayImage(); // үүсгэх хэрэггүй
image1.displayImage(); // үүсгэх хэрэггүй
```

гэж туршихад:

```
Loading   HiRes_10MB_Photo1
Displaying HiRes_10MB_Photo1
Displaying HiRes_10MB_Photo1
Loading   HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo1
```

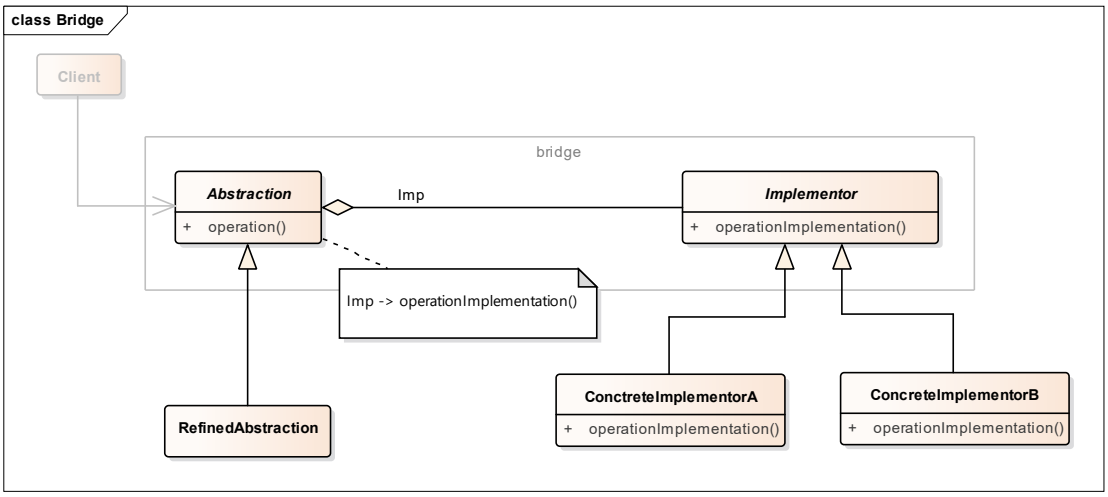
гэсэн үр дүн гарна. Эндээс объектыг үүсгэсний дараа дахин үүсгэхгүй байгааг анзаарч болно.

- **Protection proxy** нь эх объектод хандах эрхийг удирддаг. Энэ proxy нь объектууд өөр өөр хандах эрхтэй байхад хэрэг болдог.
- **Smart proxy** нь объектод хандахад нэмэлт үйлдэл хийхээр хавчуулж өгдөг. Нийтлэг ашиглалт гэвэл:
    - бодит объектыг хэдэн удаа референц хийж байгааг тоолсноор референц байхгүй үед автоматаар объектыг чөлөөлдөг.
    - байнгын объектыг анх референц хийхэд санах ойд ачаалж өгөх.
    - хандахаас өмнө бодит объектыг түгжээтэй буюу бусад объект өөрчилж чадахгүйг шалгах.

BRIDGE

Хийсвэрлэлтийг хэрэгжүүлэлтээс нь салгаж эдгээр хоёрыг нэгнээсээ хамааралгүйгээр ялгагддаг болгох.

Хийсвэрлэлийг хэд хэдэн хэрэгжүүлэлттэй болгохын тулд ихэвчлэн удамшил ашигладаг. Хийсвэрлэлтэнд хийсвэр класс интерфейсийг тодорхойлдог. Харин бодит дэд классууд олон төрлөөр хэрэгжүүлдэг. Гэхдээ энэ хандлага нь уян хатан биш. Удамшил нь хэрэгжүүлэлтийг хийсвэрлэлтэнд бүр мөсөн уяж өгдөг. Үүний үр дүнд өөрчлөх, дэлгэрүүлэх, тус тусад нь дахин ашиглахад хэцүү болдог.

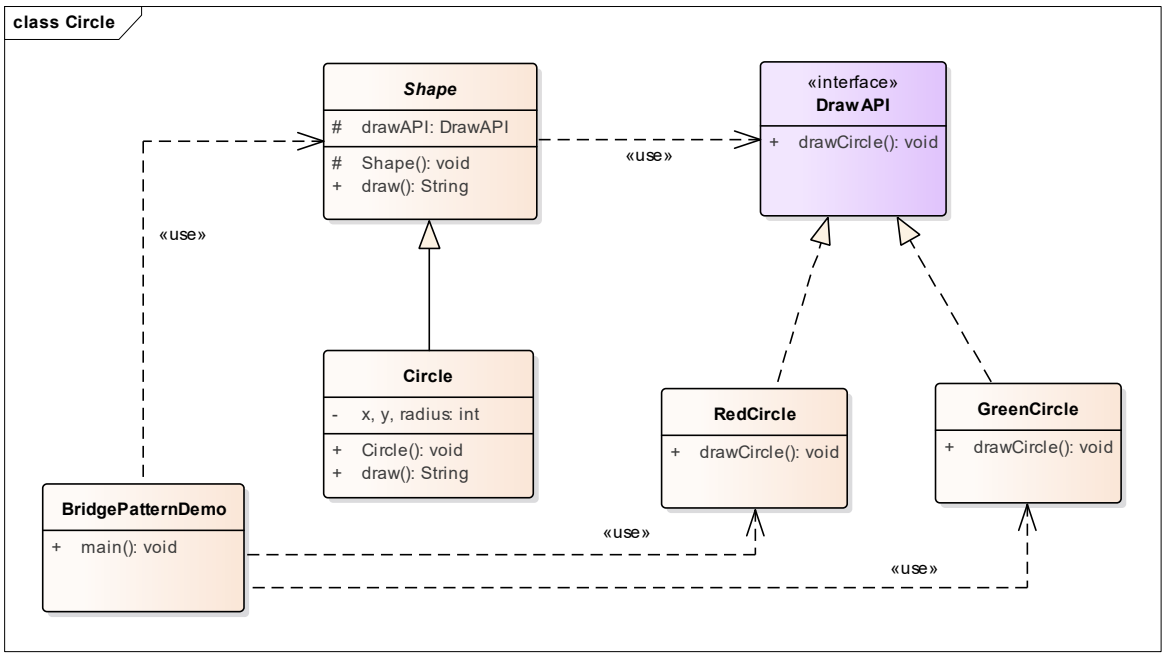


Диаграм 11

Bridge паттерн нь энэхүү асуудлыг хийсвэрлэл болон хэрэгжүүлэлтийг тус тусын шаталсан классын бүтэц болгож шийддэг.

- Abstraction – хийсвэрлэлийн интерфейсийн тодорхойлно.
- RefinedAbstraction – Abstraction-ийн тодорхойлсон интерфейсийн дэлгэрүүлэг
- Implementor – Хэрэгжүүлэх классуудын интерфейсийн тодорхойлолт
- ConcreteImplementor– Implementor интерфейсийн бодит хэрэгжүүлэлт

Дээрх bridge pattern дээр бүтээгдсэн дараах жишээг авч үзье.



Диаграм 12

Bridge хэрэгжүүлэлтийн интерфейс:

```
public interface DrawAPI {public void drawCircle(int radius, int x, int y);}
```

DrawAPI интерфейсийг хэрэгжүүлэх бодит bridge хэрэгжүүлэгч классууд:

```
public class GreenCircle implements DrawAPI {
	@Override
	public void drawCircle(int radius, int x, int y) {
		System.out.println("Drawing Circle[ color: green, radius: " + radius + ",
x: " + x + ", " + y + "]");
	}
}

public class RedCircle implements DrawAPI {
	@Override
	public void drawCircle(int radius, int x, int y) {
		System.out.println("Drawing Circle[ color: red, radius: " + radius + ",
x: " + x + ", " + y + "]");
	}
}
```

DrawAPI интерфейсийг ашиглан бүтээсэн Shape хийсвэр класс:

```
public abstract class Shape {

	protected DrawAPI drawAPI;

	protected Shape(DrawAPI drawAPI) {
		this.drawAPI = drawAPI;
	}

	public abstract void draw();
}
```

Shape интерфейсийг хэрэгжүүлсэн бодит класс:

```
public class Circle extends Shape {

	private int x, y, radius;

	public Circle(int x, int y, int radius, DrawAPI drawAPI) {
		super(drawAPI);
		this.x = x;
		this.y = y;
		this.radius = radius;
	}

	public void draw() {
		drawAPI.drawCircle(radius, x, y);
	}
}
```

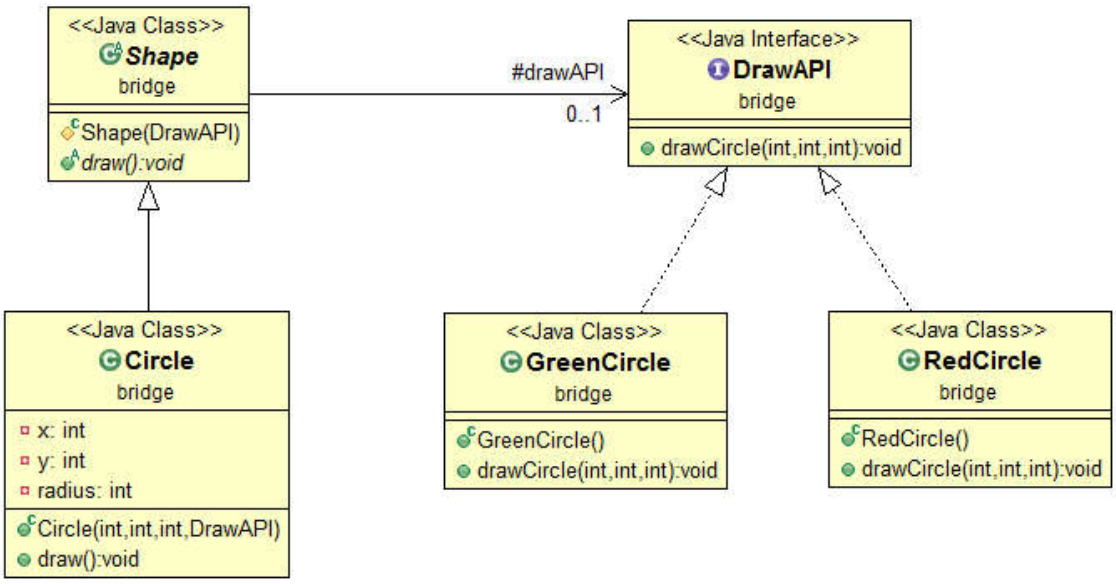
Програмыг туршиж үзэх клиент класс:

```
public class BridgePatternDemo {

	public static void main(String[] args) {
		Shape redCircle = new Circle(100, 100, 10, new RedCircle());
		Shape greenCircle = new Circle(100, 100, 10, new GreenCircle());

		redCircle.draw();
		greenCircle.draw();
	}
}
```

Үр дүн нь:

```
Drawing Circle[ color: red, radius: 10, x: 100, 100]
Drawing Circle[ color: green, radius: 10, x: 100, 100]
```

Клиентийн ашиглаж болох “handle” объект; хэрэгжүүлэлтийн объект “body” хоёр нь тусдаа өөрчлөгдөж болохоор хайрцаглагдсан байгааг дээрхээс харж болно.

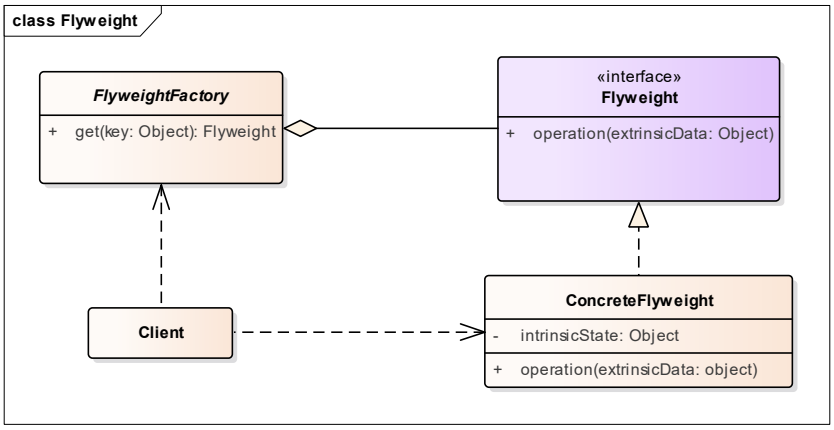


Диаграм 13

FLYWEIGHT

Хуваан хэрэглэлт ашиглан нарийн ширхэг бүхий объектуудыг үр дүнтэй дэмжих зорилттой юм. Flyweight зохиомжийн паттерн нь классын маш олон объект үүсгэх шаардлагатай үед ашигладаг юм.

Flyweight паттернийг ашиглахын өмнө объектын шинж чанарыг **intrinsic** болон **extrinsic** болгож хуваах шаардлагатай. Intrinsic шинж чанар нь объектын онцгой шинж болдог. Харин extrinsic шинж чанар нь клиентийн зүгээс олгогддог бөгөөд үйлдэл хийхэд хэрэглэгддэг байна. Жишээ нь Тойрог объект өнгө, өргөн гэсэн extrinsic шинжтэй байж болно.



Диаграм 14

Flyweight паттернийг хэрэгжүүлэхийн тулд хуваалцдаг объектыг буцаадаг Flyweight factory үүсгэх хэрэгтэй. Flyweight factory-г клиент програм объектыг жишээгээр үүсгэхэд ашигладаг. Тиймээс клиент хандаж болохгүй объектын газрын зургийг factory-д хадгалах хэрэгтэй.

```
class FlyweightFactory {
    private Map<Integer, ConcreteFlyweight> flyweights = new HashMap<Integer,
ConcreteFlyweight();

    public ConcreteFlyweight get(Integer key) {
        ConcreteFlyweight flyweight = flyweights.get(key);

        if (flyweight == null) {
            flyweight = new ConcreteFlyweight(key);
            flyweights.put(key, flyweight);
        }

        return flyweight;
    }
}
```

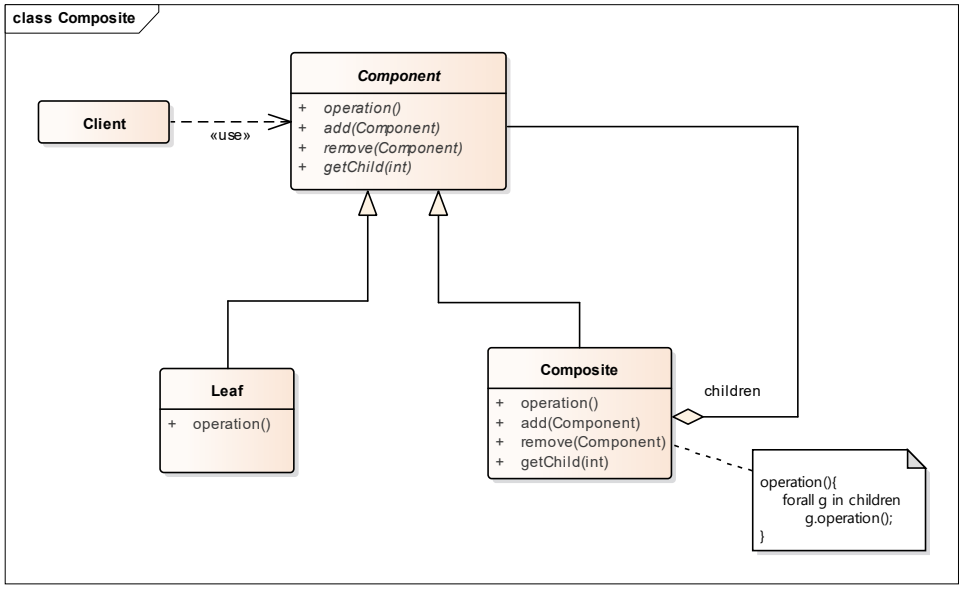
Дээрх FlyweightFactory классыг авч үзье. HashMap дээр flyweight объектуудыг хадгалж байгааг харж болно. Дараагаар нь тодорхой түлхүүр бүхий flyweight үүссэн эсэхийг шалгаад үүсээгүй бол шинээр үүсгээд HashMap-т хийнэ. Харин flyweight үүссэн байгаа бол түүнийг нь буцаана.

COMPOSITE

Composite паттерн нь объектуудыг хэдэн хэсэг нийлж бүхэл болдог шаталсан бүтцийг төлөөлөх модон бүтэц болгон цэгцэлнэ. Composite нь тусдаа объектууд болон объектуудын бүрэлдэхүүнүүдрүү клиентэд нэгэн ижил хэлбэрээр харьцах боломж олгодог.

Зураг янзалдаг эсвэл диаграм зурдаг гэх мэт зурагтай ажилладаг програм нь комплекс диаграмыг нэг бүрэлдэхүүн хэсгээс үүсгэх боломжийг хэрэглэгчид олгодог. Хэрэглэгч бүрэлдэхүүн хэсгүүдийг нэгтгээд нэг том бүрэлдэхүүн хэсэг болгоод түүнийгээ дахин нэгтгээд бүүр том бүрэлдэхүүн хэсэг болгож чадна. Энгийн хэрэгжүүлэлт гэвэл бичвэр, зураас, дөрвөлжин гэх мэт үндсэн дүрсүүд болон эдгээрийг агуулах классуудыг тодорхойлж өгөх юм.

Гэхдээ энэ аргаар хийхэд асуудал байна: Эдгээр классыг ашигладаг код дүрс болон агуулагч объекттой хэрэглэгч ихэвчлэн адилхан харьцдаг ч гэсэн өөрөөр харьцах ёстой. Эдгээр объектыг ялгах гэсээр байгаад програм хэт төвөгтэй болдог. Composite паттерн нь яаж рекурсив бүрэлдэхүүн ашигласнаар клиент нь дээрх ялгалыг хийхгүй болгохыг тайлбарладаг.

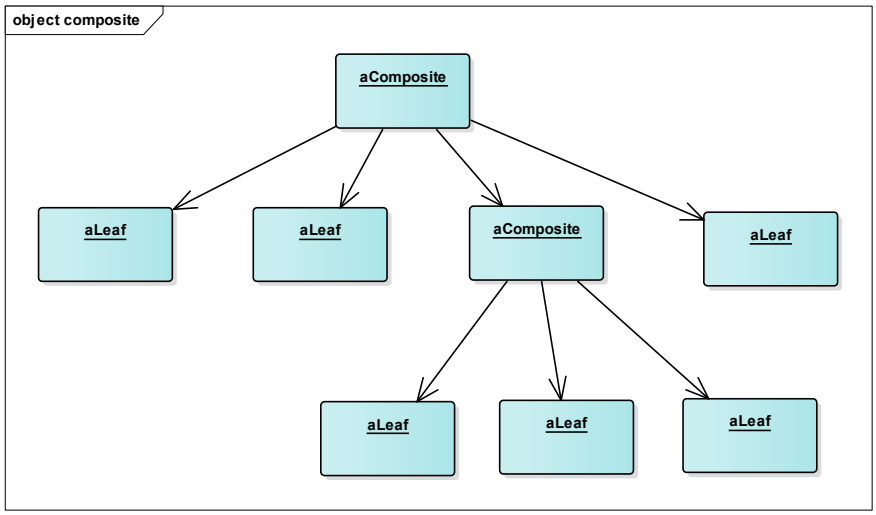


Диаграм 15

- Component
  - бүрэлдэхүүн болж буй объектын интерфейсийг тодорхойлно.
  - child объектуудыг зохицуулах, хандах интерфейсийг тодорхойлно.
- Leaf
  - бүрэлдэхүүний навч болох объектыг төлөөлнө. Навч нь хүүгүй.
  - бүрэлдэхүүний энгийн объектын шинжийг тодорхойлно.
- Composite
  - хүүтэй бүрэлдэхүүн хэсгүүдийн шинжийг тодорхойлно.

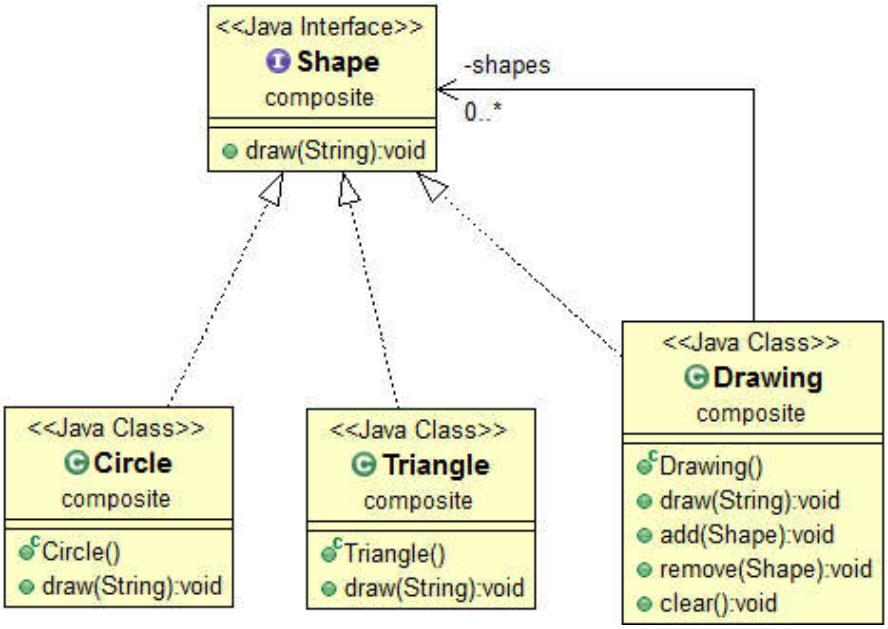
    - хүү бүрэлдэхүүн хэсгийг хадгална
    - Component интерфейсд хүүтэй-холбоотой үйлдлүүдийг хэрэгжүүлнэ.
- Client
    - composition дахь объектуудыг Component интерфейсээр ашиглана.

Ердийн Composite объектын бүтэц нь доорхтой адил харагдана:



Диаграм 16

Composite паттерн ашигласан дараах жишээг авч үзье:



Диаграм 17

Эндээс Shape component бүхий Circle болон Triangle класс нь leaf, Drawing нь composite класс болохыг шууд харж болно.

Composite класс:

```
public class Drawing implements Shape{

	//Shape-ийн цуглуулга
	private List<Shape> shapes = new ArrayList<Shape>();

	@Override
	public void draw(String fillColor) {
		for(Shape sh : shapes)
		{
			sh.draw(fillColor);
		}
	}

	//Drawing-т Shape нэмнэ.
	public void add(Shape s){
		this.shapes.add(s);
	}

	//Drawing-аас Shape-ийг хасна
	public void remove(Shape s){
		shapes.remove(s);
	}

	//Бүх Shape-ийг Drawing-аас хасна.
	public void clear(){
		System.out.println("Clearing all the shapes from drawing");
		this.shapes.clear();
	}
}
```

Leaf класс:

```
public class Circle implements Shape {
	@Override
	public void draw(String fillColor) {
		System.out.println("Drawing Circle with color "+fillColor);
	}
}

public class Triangle implements Shape {
	@Override
	public void draw(String fillColor) {
		System.out.println("Drawing Triangle with color "+fillColor);
	}
}
```

Эндээс composite класс leaf-тэй адил шинжтэй боловч leaf элементийг дотроо агуулж буйг харж болно. Туршилт хийж үзэхэд:

```
Shape tri = new Triangle();
Shape tri1 = new Triangle();
Shape cir = new Circle();

Drawing drawing = new Drawing();
drawing.add(tri1);
drawing.add(tri1);
drawing.add(cir);

drawing.draw("Red");
```

гэхэд drawing-т буй бүх объектыг улаан өнгөтэй гэж хэвлэх болно.

Гарах үр дүн:

```
Drawing Triangle with color Red
Drawing Triangle with color Red
Drawing Circle with color Red
```

`drawing.clear();` гэж цэвэрлэхэд “`Clearing all the shapes from drawing`” гэсэн үр дүн хэвлэнэ.

```
drawing.add(tri);
drawing.add(cir);
drawing.draw("Green");
```

гэхэд drawing-т байгаа 2 объектыг ногоон өнгөтэй гэж хэвлэнэ.

Гарах үр дүн:

```
Drawing Triangle with color Green
Drawing Circle with color Green
```

BEHAVIORAL ЗОХИОМЖИЙН ПАТТЕРН

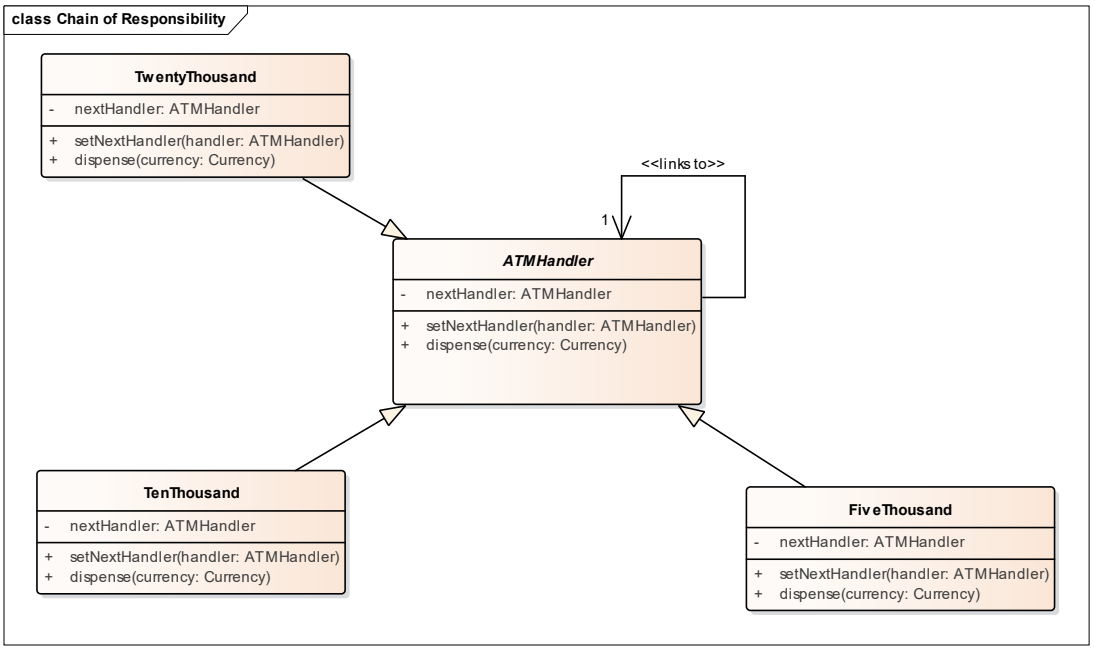
Behavioral паттерн нь объект хоорондын хариуцлага олголттой хамааралтай юм. Энэ паттерн нь зөвхөн объект болон классын паттерныг тайлбарлахаас гадна хэрхэн хоорондоо харилцах паттерныг тайлбарлана. Эдгээр паттерн нь програм ажиллах үеэр дагахад хэцүү комплекс удирдлагын урсгалыг дүрсэлдэг. Энэ Паттерн нь таны анхаарлыг удирдлагын урсгалаас холдуулж зөвхөн объект хоорондоо ямар холбоотой байгаад төвлөрөх боломж олгодог.

Behavioral нийтлэг паттерн гэвэл:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template
- Visitor

CHAIN OF RESPONSIBILITY

Нэгээс олон клиентийн хүсэлтийг биелүүлэх, хариуцах объект байхад Chain of Responsibility (CoR) паттерн нь эдгээр объект бүрт дэс дараатайгаар хүсэлтийг боловсруулахыг санал болгодог. CoR паттерныг ашигласнаар, эдгээр бүх боловсруулагч объектийг нэг нь нөгөөхөө заасан заагч бүхий гинж хэлбэрээр цэгцлэх боломжтой. Эхний объект хүсэлт авахдаа цааш дамжуулах эсвэл өөрөө боловсруулахаа шийднэ. Энэ мэтээр хүсэлт бүх объектоор дамжихад аль нэг объект нь хүсэлтийг боловсруулдаг эсвэл хүсэлт ердөөсөө боловсруулагдаагүй гинжийн эцэс хүрэх хүртэл үргэлжилнэ.



Дээрх диаграмт CoR хамгийн нийтлэг жишээ ATM-ийг харуулсан байна. ATM нь 20, 10, 5 мянга төгрөг ихээс бага гэсэн дараалалтайгаар гаргадаг билээ. Хэрвээ дээрх диаграмын хэрэгжүүлэлтийг харвал:

Currency буюу мөнгөн дүнг хадгалах класс.

```
public class Currency {

	private int amount;

	public Currency(int amt) {
		this.amount = amt;
	}

	public int getAmount() {
		return this.amount;
	}
}
```

ATMHandler буюу хүсэлтийг зохицуулах интерфейс бүхий хийсвэр класс.

```
public abstract class ATMHandler {
		private ATMHandler nextHandler;

		public abstract void dispense(Currency cur);

		public ATMHandler getNextHandler() {
			return nextHandler;
		}

		public void setNextHandler(ATMHandler handler){
			nextHandler = handler;
		}

}
```

ATMHandler-ийн TwentyThousand, TenThousand, FiveThousand гэх дэд классуудын ажиллагаа нь өөрсдийн оноогсон хэмжээнд currency гаргаж өгөөд үлдэгдэл үлдвэл цуваан дахь дараагийн класст дамжуулна.

```
public class TwentyThousand extends ATMHandler {
	@Override
	public void dispense(Currency cur) {
		if (cur.getAmount() >= 20000) {
			int num = cur.getAmount() / 20000;
			int remainder = cur.getAmount() % 20000;
			System.out.println("Dispensing " + num + " 20000 note");
			if (remainder != 0)
				this.dispense(new Currency(remainder));
		} else {
			getNextHandler().dispense(cur);
		}
	}
}
```

ATMDispenseChain Класс нь програмын ажиллагааг харуулсан байна. ATMHandler объект үүсгээд chain of responsibility үүсгэснийг `handler1.setNextHandler(handler2)` гэснээс харж болно.

```
public class ATMDispenseChain {
	private ATMHandler handler1;

	public ATMDispenseChain() {
		this.handler1 = new TwentyThousand();
		ATMHandler handler2 = new TenThousand();
		ATMHandler handler3 = new FiveThousand();

		handler1.setNextHandler(handler2);
		handler2.setNextHandler(handler3);
	}

	public static void main(String[] args) {
		ATMDispenseChain dispenseChain = new ATMDispenseChain();
		dispenseChain.handler1.dispense(new Currency(135000));
	}
}
```

```
“Dispensing 6 20000 note
Dispensing 1 10000 note
Dispensing 1 5000 note”
```

Тухайн програмыг ажиллуулахад дээрх үр дүн гарах болно.