DISEINU PATROIEN ADIBIDEAK

- ADAPTER
- STRATEGY
- FACTORY
- ABSTRACT FACTORY

EGILEA : CARLOS IRIBARREN ZUBIZARRETA

ADAPTER

Diseinu Patroi honen helburu nagusia objetu bat beste objetu bati adaptatzea da, horrela adaptatutako objetuaren funtzionalitatea, beste ingurune desberdin batean berrerabili daitekelarik.

Adibidez programa baten kodean ezin baduzu kodea aldatu baina bere funtzionalitateko gauza batzuk aprobetzatu nahi badituzu, aukera ona da adapter patroia erabiltzeko. Horrela beste erdiko klase bat sortu dezakezu aipaturiko adatazioa aprobetzatu ahal izateko.

- Adibidea :

Adibide honetan hasieran klase bakarra daukagu, batura izenekoa. Hau programa bat da non erabiltzaileari 2 zenbaki sartzeko eskatzen zaion eta hauen batura egiten da. Batura funtzio baten bidez kalkulatze da:

```
public Integer bizenbakibatu(Integer z1,Integer z2)
{
        return z1 + z2;
}
```

Eta orain programa berri bat egin behar dugu, Biderkaketa bat egiten duen programa. Biderkaketa egiteko N batura egin behar dira, hortaz baturaren funtzioa berrerabili dezakegu. Biderkaketa Interface bat definituko dugu bere metodoak beste klase batean implementatzeko.

```
public interface BiderInterface
{
         public abstract Integer bizenbakiBiderkatu(Integer z1,Integer z2);
}
```

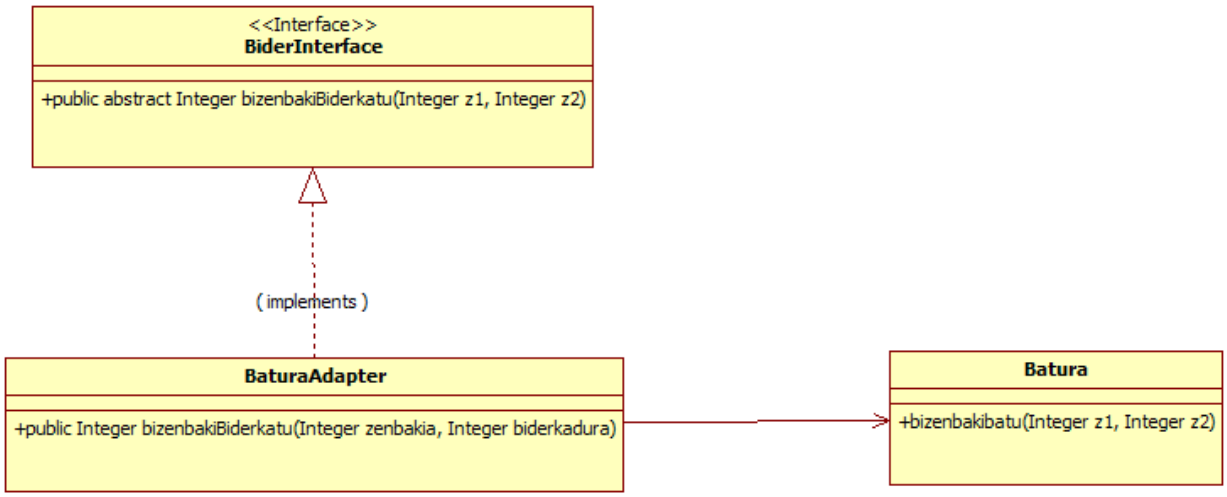
Adapter klaseak metodo hoiek implementatuko ditu eta Batura klasera deia egingo du, horrela bi klaseen arteko erlazioa sortuz eta adaptazio prozesua eginez.

```
import Adapter.LogikaMaila.Batura;
public class BaturaAdapter  implements BiderInterface {

	@Override
	public Integer bizenbakiBiderkatu(Integer zenbakia, Integer biderkadura)
	{
		Batura nereBatura = new Batura();
		 Integer emaitza = new Integer(0);
		 int a;
		for (a=0; a<biderkadura ;a++)
		{
			emaitza = nereBatura.bizenbakibatu(emaitza, zenbakia);
		}
		return emaitza;
	}
}
```

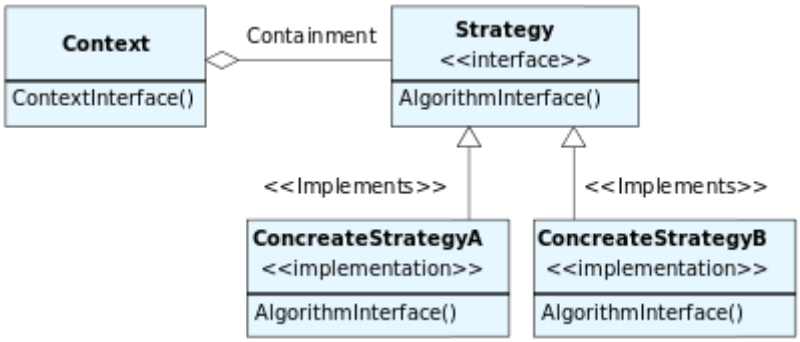
Azkenik gure programa nagusitik BaturaAdapter instantzia bat sortu eta bere metodoak erabiliko ditugu.

```
BaturaAdapter  nere_BiderkaketaAdapter = new BaturaAdapter();
erantzuna = nere_BiderkaketaAdapter.bizenbakiBiderkatu(z1, z2);
```



STRATEGY

Diseinu Patroi honen helburu nagusia erlazionaturik dauden algoritmoak klaseetan enkapsulatzea eta trukakorrak izatea da. Horrela algoritmoak dinamikoki aldatu daitezke eta “if” eta “swich” moduko aginduak desagertu egiten dira.



- Adibidea :

Adibide honetan Bideojoko baten mailak implementatuko ditugu. Adibidez lehengo pantailarako 2 maila dauden, zailtasunaren arabera. 1.maila ezaugarri batzuk ditu eta 2.maila ezaugarri desberdinak, hau da, zailagoa da 2.maila (abiadura azkarragoa,distantzia handiagoa).

Horretarako Joko klasea definituko dugu Abstract edo Interface modukoa, gero bere metodoak beste klasetan definitzeko.

```
public abstract class Jokoa
{
        private Integer maila;
        private Integer metroak;

        public abstract void HasiJokoa();

        ...
}
```

Ondoren JokoaMaila1 sortuko dugu eta extends edo implements (abstract edo interface arabera) egingo du Jokoa klasetik.

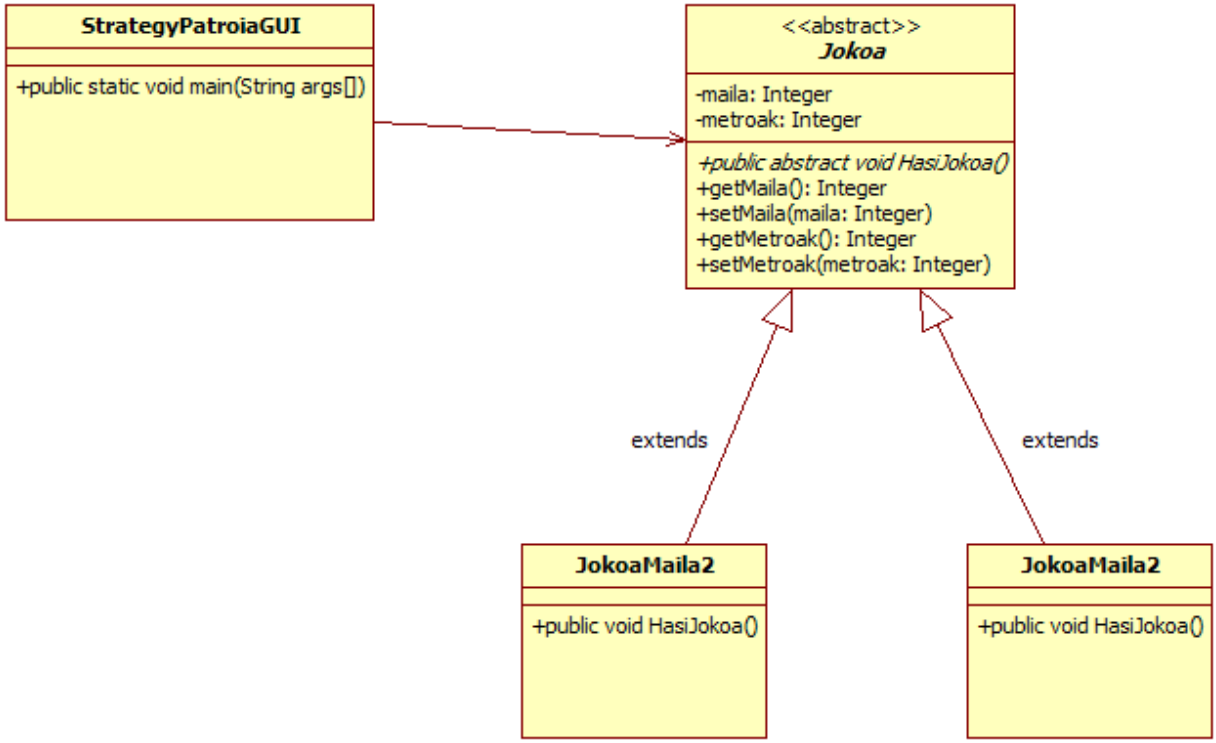
```
public class JokoaMaila1 extends Jokoa
{
         @Override
        public void HasiJokoa()
        {
                this.setMaila(1);
                this.setMetroak(2500);
        }
 }
```

Eta berdina JokoaMaila2 bere ezaugarri bereziekin

```
public class JokoaMaila2 extends Jokoa
{
        @Override
        public void HasiJokoa()
        {
                this.setMaila(2);
                this.setMetroak(6000);
        }
 }
```

Azkenik gure programa nagusitik dagokion JokoaMailaX instantzia sortu eta bere metodoak erabiliko ditugu.

```
nere_Jokoa_Maila1 = new JokoaMaila1();
nere_Jokoa_Maila1.HasiJokoa();
```



FACTORY

Diseinu Patroi honen helburu nagusia Objetuak sortzeko fabrika bat implementatzea da. Objetuak dinamikoki sortuko dira erabiltzailearen beharrei erantzuna emanez. Objetuak desberdinak izango dira baina familia berekoak.

- Adibidea :

Adibide honetan supermerkatuko objetuak sortzeko Fabrika implementatuko dugu. Hasieran fruta klase abstraktu bat definituko da. Fruta guztien ezaugarri komunekin.

```
public abstract class  FrutaAbstract
{
	private String izena;
	private double prezioKilo;
	private String denboraldia;

	...
}
```

Ondoren fruta bakoitzaren implementazioa egingo da Extennds eginez.

```
public class Sagarra extends FrutaAbstract {

	public Sagarra()
	{
		this.setDenboraldia("Uda hasiera");
		this.setIzena("Sagarra");
		this.setPrezioKilo(2.78);
	}
 }

public class Limoia extends FrutaAbstract {

	public Limoia()
	{
		this.setIzena("Limoia");
		this.setDenboraldia("Uda");
		this.setPrezioKilo(2.0);
	}
 }

public class Laranja extends FrutaAbstract {

	public Laranja()
	{
		this.setIzena("Laranja");
		this.setPrezioKilo(2.5);
		this.setDenboraldia("Udaberria");
	}
}
```

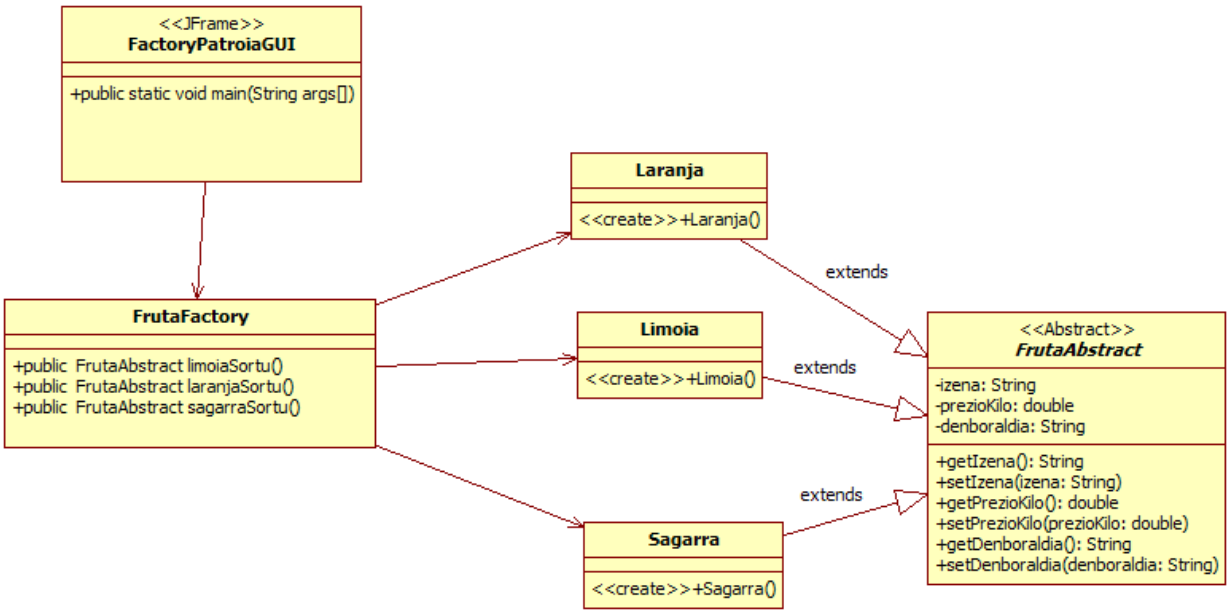
Horrela gure objetu bilduma definituta eukiko dugu. Orain (Fruta) Objetu Fabrika implementatuko da. Hemen metodo bat egongo da produktu (fruta) bakoitzeko, eta honek dagokion produktua (fruta) hitzuliko du exekuzio-denboran.

```
public class FrutaFactory
{
	public  FrutaAbstract limoiaSortu()
	{
		return new Limoia();
	}

	public  FrutaAbstract laranjaSortu()
	{
		return new Laranja();
	}

	public  FrutaAbstract sagarraSortu()
	{
		return new Sagarra();
	}

}
```



ABSTRACT FACTORY

Diseinu Patroi hau Factory patroiaren hedapen bat da. Helburu nagusia Fabrikak sortzeko Fabrika Abstraktu bat implementatzea da. Fabrika Abtraktuaren metodoen implementazioaren ardura Fabrika subKlasei (semei) uzten delarik. Horrela produktuak eta bezeroak guztik independienteak dira.

- Adibidea :

Adibide honetan Frabrika Abstraktua implementatuko da. Produktuen implementazio osoa Factory adibidean azalduta dago, hortaz, ez da hemen berriz azalduko. Hasieran Fabrika Abstraktua definituko dugu eta metodoen implementazioa Fabrika bakoitzean definituko delarik. Metodoak hitzuliko duen datu mota Objetk motakoa da, jasoko dugun produktua familia desberdinekoa izan daiteleelako.

```
public abstract class FrutaFabrikaNagusiaAbstract
{

	public abstract Object sortuLimoia();
	public abstract Object sortuLaranja();
	public abstract Object sortuSagarra();

}
```

Ondoren Fabrika konkretu bakoitza definituko da, metodoen implementazioa eginez.

```
public class FrutaBizkaiaFactory extends FrutaFabrikaNagusiaAbstract
{
	@Override
	 public FrutaBizkaiaAbstract sortuLimoia()
	{
		 return new BizkaiaLimoia();
	}
	@Override
	public FrutaBizkaiaAbstract sortuLaranja()
	{
		return new BizkaiaLaranja();
	}
	@Override
	public FrutaBizkaiaAbstract sortuSagarra()
	{
		return new BizkaiaSagarra();
	}

}
```

```
public class FrutaGipuzkoaFactory extends FrutaFabrikaNagusiaAbstract
{
	@Override
	public  FrutaGipuzkoaAbstract sortuLimoia()
	{
		return new GipuzkoaLimoia();
	}
	@Override
	public  FrutaGipuzkoaAbstract sortuLaranja()
	{
		return new GipuzkoaLaranja();
	}
	@Override
	public  FrutaGipuzkoaAbstract sortuSagarra()
	{
		return new GipuzkoaSagarra();
	}

}
```

Azkenik programa nagusitik behar dugun Fabrika eta Produktua sortuko da. “Object” datu mota hitzultzen duenez dagokion Cast-ina egin beharko da.

```
FrutaGipuzkoaFactory nereFrutaFactory = new FrutaGipuzkoaFactory();
FrutaGipuzkoaAbstract nereFruta;
nereFruta = (FrutaGipuzkoaAbstract) nereFrutaFactory.sortuLaranja();
```