

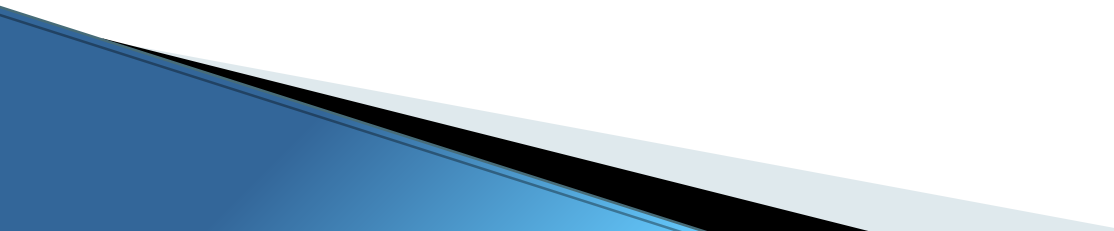
Herentzia anizkoitza

Java Interfazeak

SOFTWARE INGENIARITZA



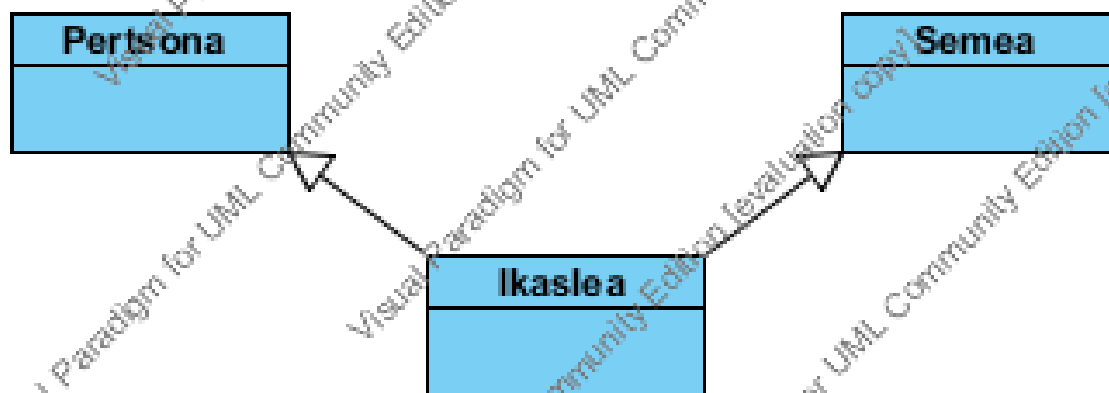
EDUKIAK

- ▶ Motibazioa
 - ▶ Abantailak
 - ▶ Diamantearen arazoa
 - ▶ Interfazeak
- 

Motibazioa

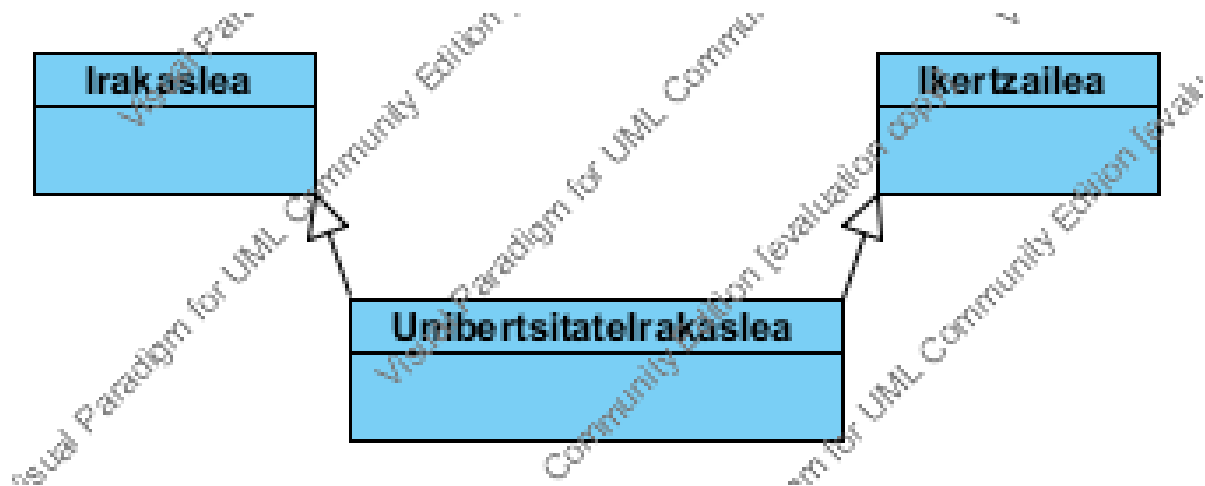
Mundu errealean entitateek guraso bat baino gehiagotik heredatzen dituzte ezaugarriak eta jokabideak.

Herentzia anizkoitza objektuei bideratutako programazio lengoiaen mekanismo bat da; hots, klase batek superklase bat baino gehiagotik jokabide eta ezaugarriak heredatzea.



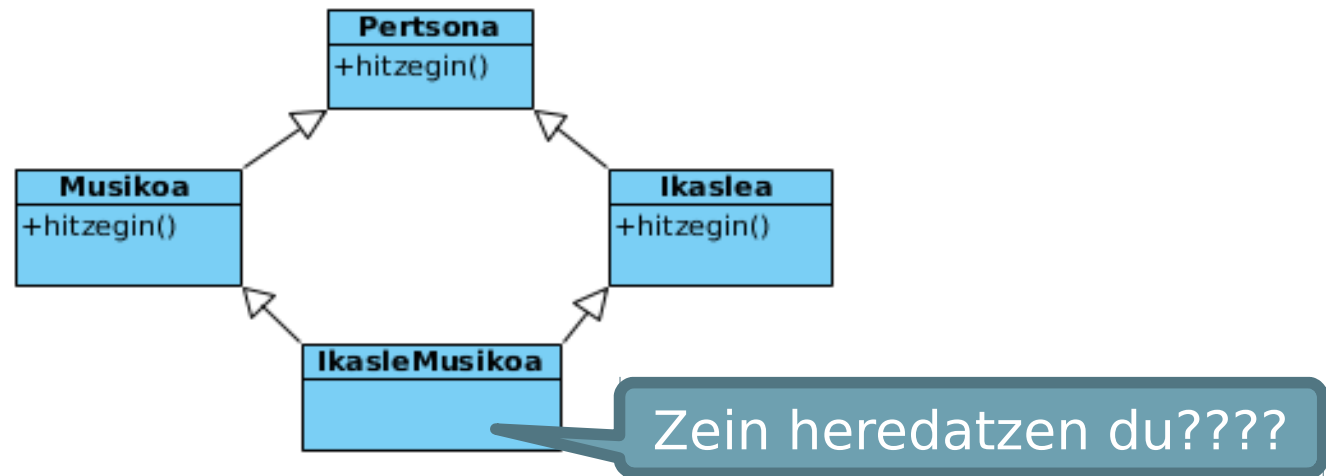
Abantailak

- Errealistagoaz gain, herentzia sinplearen berrerabilpen-aukerak handitzen ditu.



Diamantearen arazoa

Klase batek izen bereko ezaugarri edo funtzionalitate bat bi superklase ezberdinetatik heredatzen duenean.



JAVAK ez du herentzia anizkoitza onartzen!!!

Interfazeak

- JAVA interfazea klase abstraktu hutsa da
 - Metodo guztiak abstraktuak
- Atributuak eduki ditzake
 - Beti *static* and *final* izango dira.
- Beste edozein klase bezala deklaratu, baina “class” hitz erreserbatua erabili ordez “interface”.

```
public interface ilInterfazelzena { ... }
```

Interfazeak

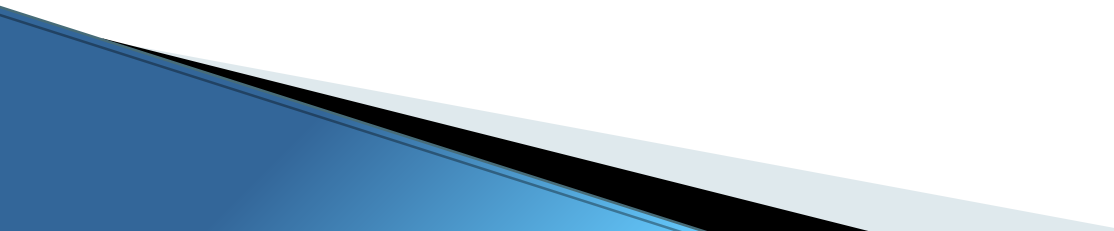
- Interfaze baten deklaraturako metodoak beti publikoak; beste klase batzuetan inplementatu.

```
public class klaselzena implements Interfazelzena  
    { ... }
```

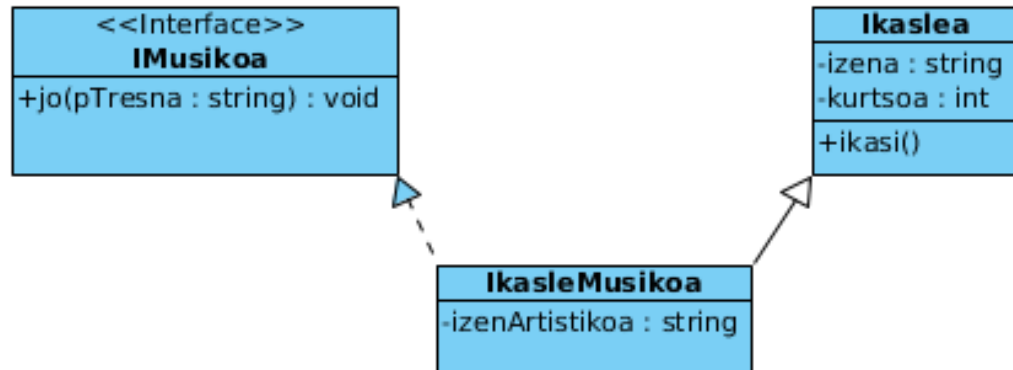
- Interfazeek beste interfaze batzuk hedatu (extends) ditzakete
- Klase batek interfaze bat baino gehiago inplementatu (implements) dezake

Interfazeak

Zertarako balio dute interfazeek?

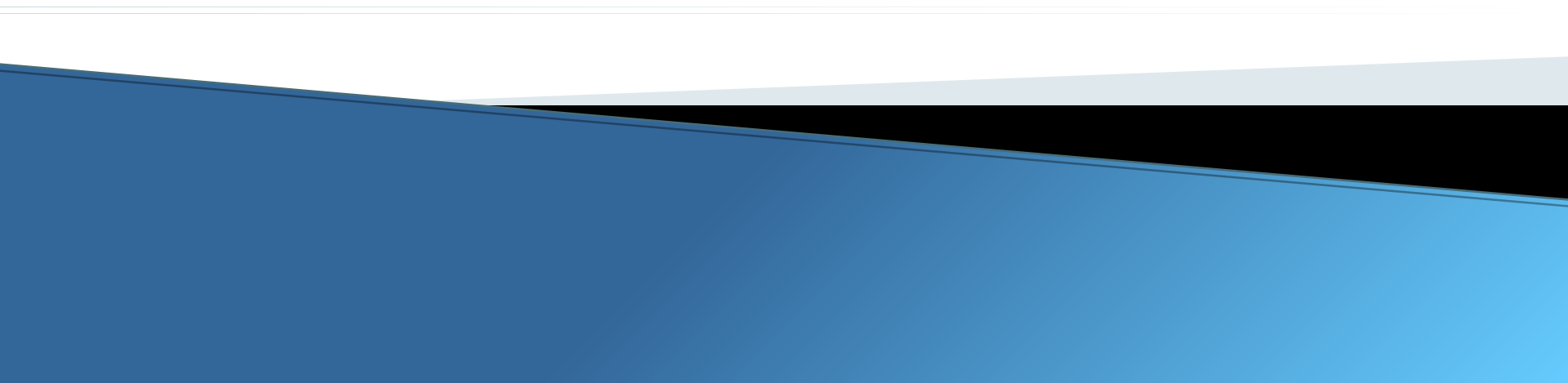
1. JAVAn herentzia anizkoitzaren gabezia betetzeko.
 2. Klase batek beste objektu batzuek eskainitako datu eta metodoak erabili ahal izateko, horien superklase konkretuegatik arduratu barik.
 3. Klaseen ezaugarri sematikoak definitzeko.
- 

Interfazeak



```
public class IkasleMusikoa extends Ikaslea
implements IMusikoa {
    public String izenArtistikoa;
    public void jo() {
        ...
    }
}
```

SOLID Printzipioak eta Diseinu Patroiak



Sarrera - Aldaketa

“Software sistemek bizi zikloan zehar aldaketak jasaten dituzte”

- Diseinu onei eta txarrei gertatzen zaie
- Diseinu onak egonkorak dira

Berdin da non zauden lanean, zer eraikitzen ari zaren edo zein programazio lengoaia erabiltzen ari zaren, beti egongo da konstante bat, **aldaketa**.

Zure aplikazioa oso ondo diseinatzen baduzu ere, denbora pasa ala aplikazioa hazi egin beharko da edo **aldaketak** jasan beharko ditu, bestela zaharkituta geldituko da.

Konstantea den gauza bakarra **aldaketa** da.

SOLID Printzipioak

- **S**ingle-Responsability Principle (SRP)
- **O**pen-Close Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

Irakurketa:

<http://mundogeek.net/archivos/2011/06/09/principios-solid-de-l-a-orientacion-a-objetos/>

Single Responsibility Principle (SRP)



Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

Irudiak hemenetik hartuta daude:

<http://blogs.msdn.com/b/cdndevs/archive/2009/07/15/the-solid-principles-explained-with-motivational-posters.aspx>

SRP

Moduluek eta klaseek SWeko **funtzionalitate bakarraren ardura** izan behar dute

SRP Adibidea

```
public class CurrencyConverter {  
  
    public BigDecimal convert(Currency from, Currency to, BigDecimal amount) {  
        // gets connection to some online service and asks it to convert currency  
        // parses the answer and returns results  
    }  
  
    public BigDecimal getInflationIndex(Currency currency, Date from, Date to) {  
        // gets connection to some online service to get data about  
        // currency inflation for specified period  
    }  
}
```

Zergatik kalkulatzeko
da inflazioa txanpon
trukaketarekin?

Eta txanpon trukaketa
zerbitzua aldatuko
balitz? Edo inflazioa
kalkulatzeko formatua?

**Ez da intuitiboa!
Gainkargatuta dago!**

**Klasea bi kasuetan
aldatu behar da!**

Konpondu!!!

SRP Adibidea

```
public class CurrencyConverter {  
    public BigDecimal convert(Currency from, Currency to, BigDecimal amount) {  
        // gets connection to some online service and asks it to convert currency  
        // parses the answer and returns results  
    }  
}
```

Inflazioa kalkulatzeko
formatua aldatzen bada?
InflationIndexCounter
bakarrik aldatzen dugu!

Txanpon trukaketa
zerbitzua aldatzen bada?
CurrencyConverter
bakarrik aldatzen dugu!

```
public class InflationIndexCounter {  
    public BigDecimal getInflationIndex(Currency currency, Date from, Date to) {  
        // gets connection to some online service to get data about  
        // currency inflation for specified period  
    }  
}
```


SRP Adibidea II

Bi ardura:
Kautotzea eta
erabiltzailea DBtik
lortzea

```
public class UserAuthenticator {  
    public boolean authenticate(String username, String password){  
        User user = getUser(username);  
        return user.getPassword().equals(password);  
    }  
}
```

```
private User getUser(String username){  
    st.executeQuery("select user.name, user.password from user where id  
    // something's here  
    return user;  
}
```

SRP Adibidea II

```
public class UserAuthenticator {  
    private UserDetailsService userDetailsService;  
    public UserAuthenticator(UserDetailsService service) {  
        userDetailsService = service;  
    }  
    public boolean authenticate(String username, String password){  
        User user = userDetailsService.getUser(username);  
        return user.getPassword().equals(password);  
    }  
}
```



Orain ez dugu zuzenean
DBarekin lan egiten



Kautoketa LDAP-era aldatuta,
Klasea ez da aldatzen

Open-Close Principle (OCP)



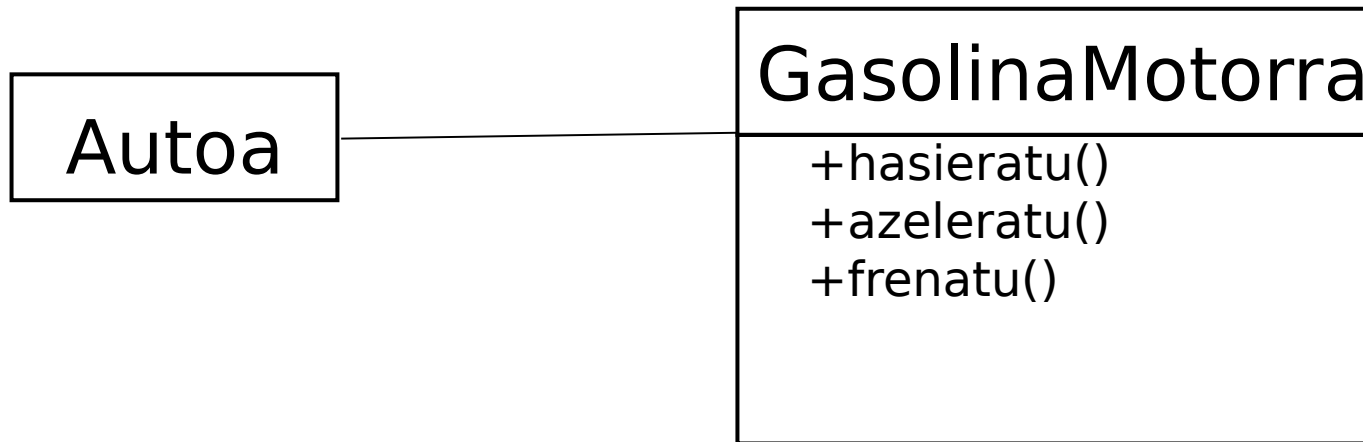
Open-Closed Principle

Open-chest surgery isn't needed when putting on a coat.

OCP

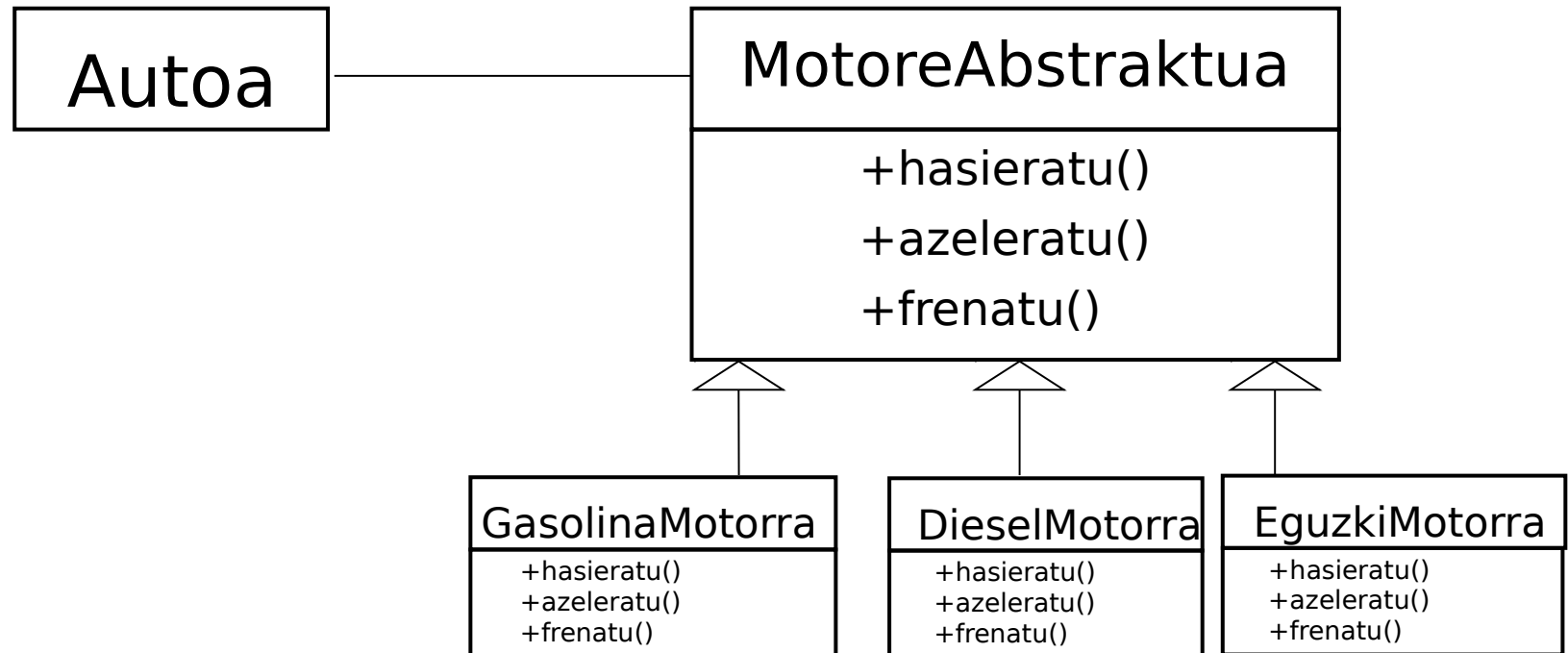
SW entitateak (funtzionalitateen) **hedapenerako irekiak**, baina (kodearen) **aldaketarako itxiak** izan behar dute

OCP Adibidea



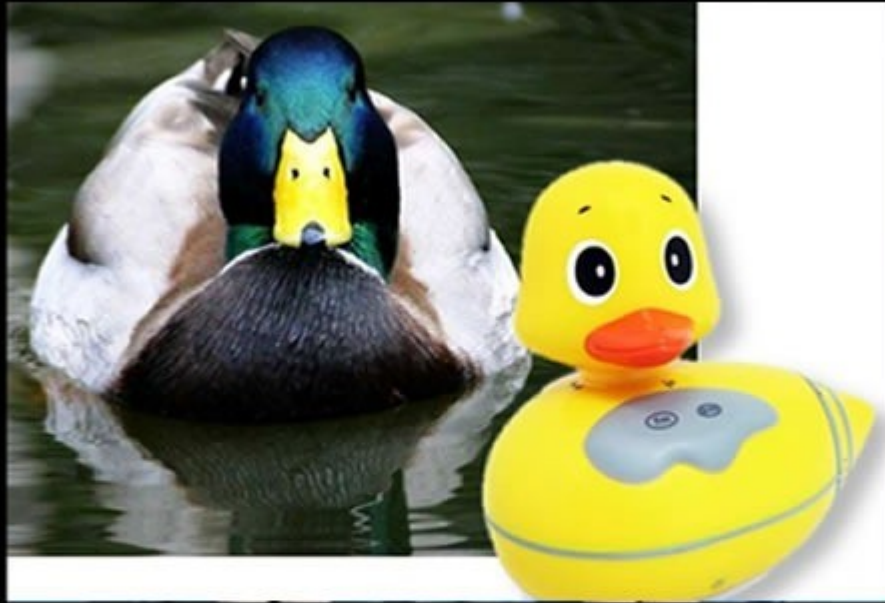
- Nola egin Auto batek *GasolinaMotorra* edo *EguzkiMotorra* erabili ditzan?
- Autoa klasea aldatu behar dugu!
 - ...gutxienez diseinu honetan

OCP Adibidea



- Ahal den elean, klase batek ez du klase konkretu batekin dependentziarik izan behar
- Klase abstraktu batekin izan behar du dependentzia.

Liskov Sustitution Principle (LSP)



Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

LSP

Klase batetik heredatzen duen azpiklase
(**seme klase**) **orok** lehenegoa (**ama klasea**) **bezala**
erabili ahal beharko litzateke, beren arteko
desberdintasunak ezagutu barik ere

LSP herentzian oinarritzen da

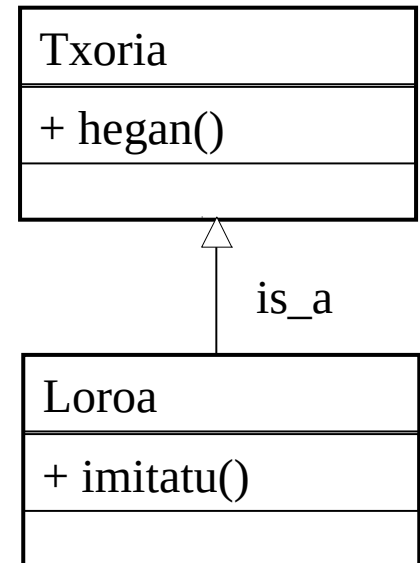
*Herentziak hurrengoa bermatu behar du: **superklasearen edozein objekturen propietate frogagarri azpiklaseen edozein objekturentzat baliogarria da.***

B. Liskov, 1987

Herentziak erraza dirudi...

```
abstract class Txoria { // lumak, hegoak... ditu
    public void hegan(); // Txoriek hegan egin dezakete
};

class Loroa extends Txoria { // Loro bat txori bat da
    public void imitatu();      // Hitzak errepikatu ditzake
};
// ...
Loroa nireMaskota=new Loroa();
nireMaskota.imitatu();        // Loroa izanda, imitatu() dezake
nireMaskota.hegan();          // Txoria izanda hegan() egin dezake
```



Pinguinoek ezin dute hegan egin

```
class Pinguinoa extends Txoria {  
    public void hegan() {  
        new Exception("ezin dut hegan egin!"); }  
};
```



```
void txoriakBezalaHegan (Txoria txori) {  
    txori.hegan(); // lora ondo  
    // Eta pinguinoa?...OOOPS!!  
}
```

- Ez du “*Pinguinoek ezin dute hegan egin*” modelatzen
- “*Pinguinoek hegan egin dezakete, baina saiatuz gero errorea*” modelatzen du
- Hegan egiten saiatzen badira → Run-time errorea
- Ordezkapen printzipioan pentsatu → Liskov printzipioa ez da betetzen

Interface Segregation Principle (ISP)



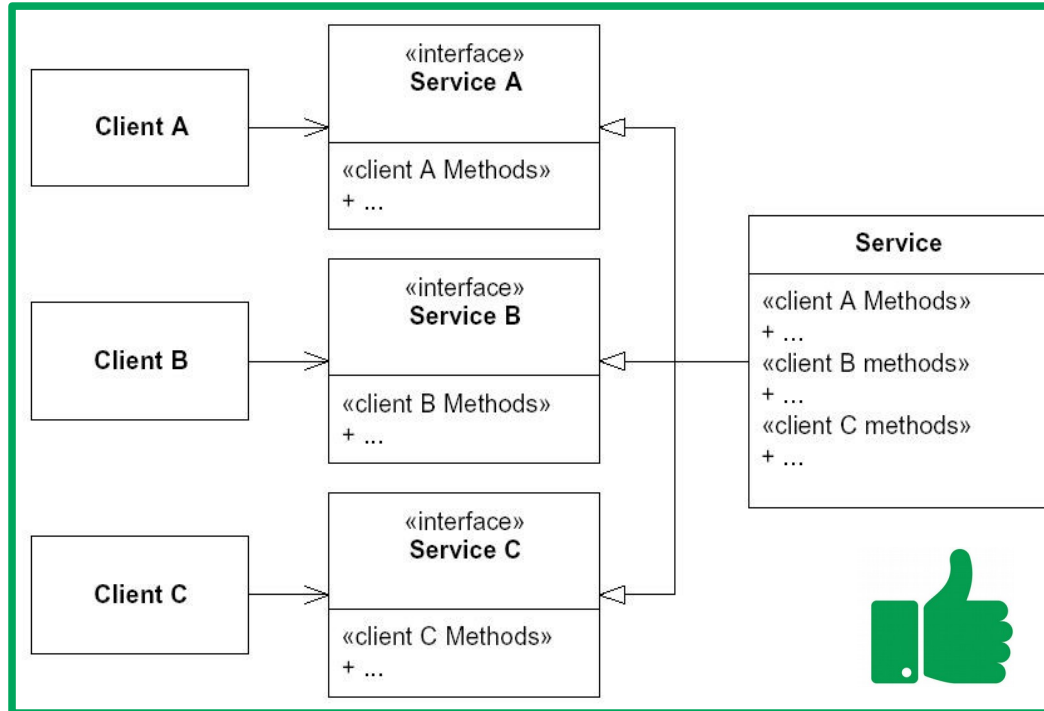
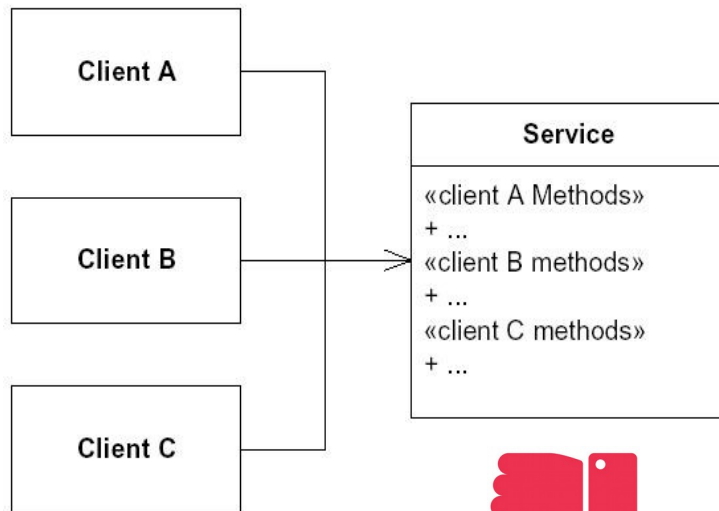
Interface Segregation Principle

You want me to plug this in *where?*

ISP

**Bezeroek, erabiltzen ez duten metodoen
dependentziarik ez dute izan behar**

Adibidea



Dependency Inversion Principle (DIP)



Dependency Inversion Principle

Would you solder a lamp directly
to the electrical wiring in a wall?

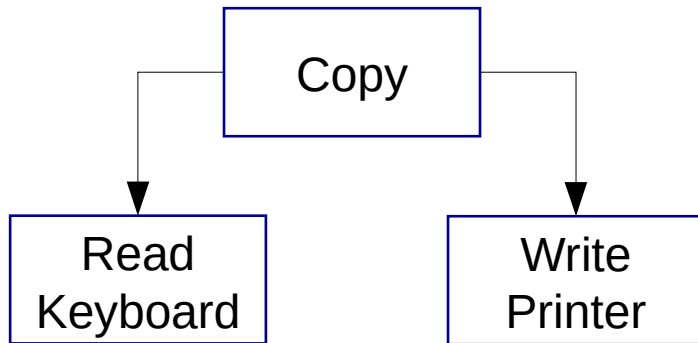
DIP

- I. **Goi mailako moduluek** ezin dute behe mailako moduluekin menpekotasunik izan. Goikoek zein behekoek, **menpekotasuna abstrakzioekin**.
- II. Abstrakzioek ezin dute xehetasunekin menpekotasunik izan. **Xehetasunek abstrakzioekin menpekotasuna**.

Martin, 1996

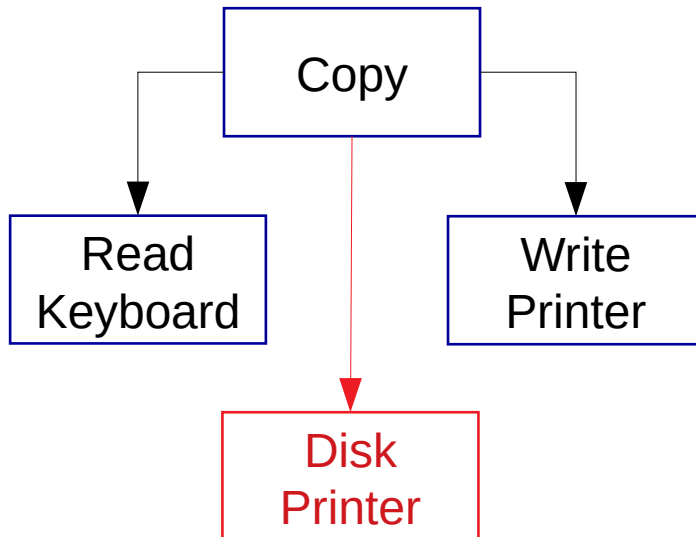
- OCPk helburua adierazten du. DIPek mekanismoa.
- Superklase batek ez ditu bere azpiklaseak ezagutu behar.
- Inplementazio xehetasunak dituzten moduluek ez dute beraien arteko menpekotasunik. Menpekotasuna abstrakzioen bidez definitzen dira.

Adibidea



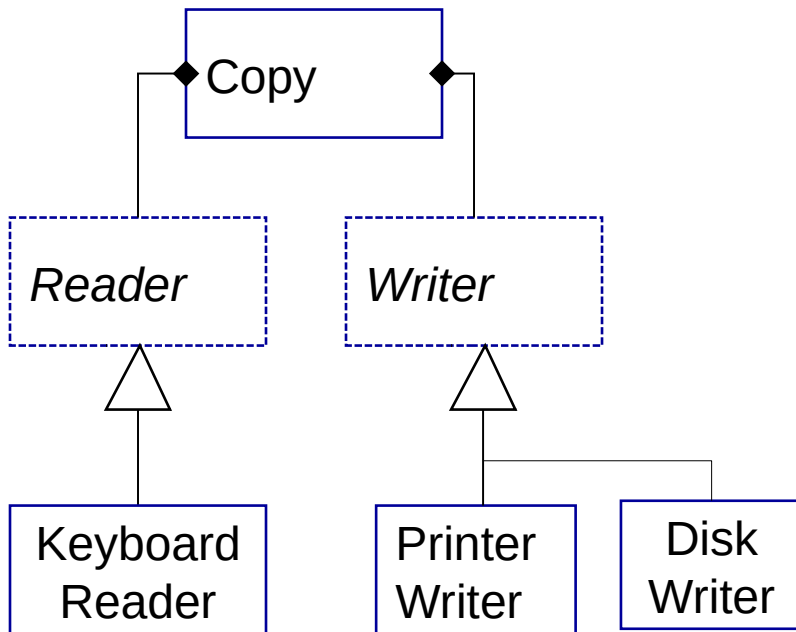
```
void Copy(){  
    int c;  
    while ((c = ReadKeyboard()) != EOF)  
        WritePrinter(c);  
}
```

Adibidea



```
enum OutputDevice {printer, disk};  
void Copy(OutputDevice dev){  
    int c;  
    while((c = ReadKeyboard())!= EOF)  
        if(dev == printer)  
            WritePrinter(c);  
        else  
            WriteDisk(c);  
}
```

Adibidea

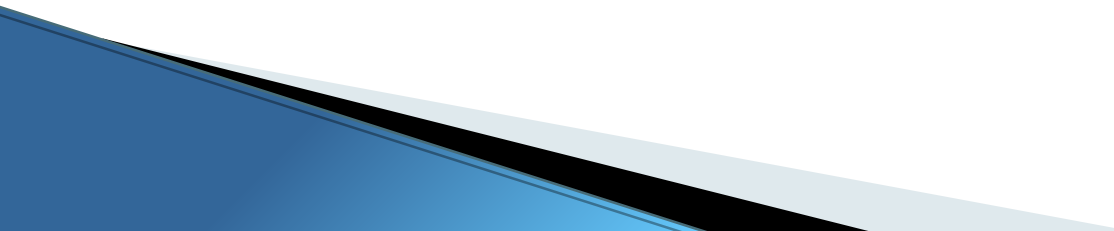


```
class Reader {
    public abstract int read(){ };
}

class Writer {
    public abstract void write(int i);
};

void copy(Reader r, Writer w){
    int c;
    while((c = r.read()) != EOF)
        w.write(c);
}
```

Laburbilduz

- ▶ SW sistemek **aldaketak** dituzte beren bizi zikloan
 - ▶ SW diseinuak aldaketetara **modatu** behar dira
 - ▶ SOLID printzipioak hastapeneko pausuak dira, diseinu konplexuak egiteko teknika sofistikatuagoak
- 

DISEINU PATROIAK

DISEINU PATROIAK

MOTIBAZIOA

- ▶ OZ diseinua, zaila!! Berrerabilgarritasuna, are zailago!!
- ▶ Iraganean funtzionatu duten soluzioak berrerabili
- ▶ PATROIEK diseinu problema zehatzak ebatzi
 - Diseinu malgua eta berrerabilgarria ahalbidetu
- ▶ SOLID printzipioetan oinarritu

DISEINU PATROIAK

DEFINIZIOA: elkarrekin komunikatzen diren objektu eta klaseen definizioa, baina, diseinu problema orokor bat testuinguru partikular batetan ebazteko egokituta

DISEINU PATROIAK

HISTORIA

- ▶ 1964-1979: Christopher Alexanderrek patroiak planteatu *arkitektura munduan*.
- ▶ 1990-1992: “Gang of Four”(GOF) taldearen lana hasi, informatikara Alexanderren ideiak.
- ▶ 1995: “***Design Patterns, Elements of Reusable Object-Oriented Software***” liburu ospetsua argitaratu

DISEINU PATROIAK

SAILKAPENA

- ▶ **Sortzaileak:** objektuen sorkuntza
- ▶ **Egiturazkoak:** klase eta objektuen konposaketa
- ▶ **Portaerazkoak:** klase eta objektuen elkarreragin eta ardurak banatzeko era

PATROIAK: ITERATOR

MOTIBAZIOA

- ▶ Objektu sortak sekuentzialki zeharkatzeko metodoak asko erabili.
- ▶ Objektu sortak era ezberdinetan implementatu daitezke.

Arrayetan oinarritutako zerrenda	Zerrenda estekatuetan oinarritutako zerrenda
<pre>public class ArrayZerrenda { private int osagaiKop; private String[] zerren; ... }</pre>	<pre>public class Nodoa { private String datua; private Nodoa hurr; ... } public class ZerrendaEstekatua{ private Nodoa lehena; ... }</pre>

PATROIAK: ITERATOR

MOTIBAZIOA

- Inplementazio bakoitzerako, osagaiak zeharkatzeko metodo desberdina.

Arrayetan oinarritutako zerrenda

```
...  
String datua = null;  
for (int i= 0; i< zerren.size();i++) {  
    datua = zerren.get(i);  
    ...  
}  
...
```

Zerrenda estekatueta oinarritutako zerrenda

```
...  
String datua = null;  
Nodoa aux = zerren.getFirst();  
while (aux!=null) {  
    datua = aux.getContent();  
    ...  
    aux = aux.getNext();  
}  
...
```

PATROIAK: ITERATOR

ARAZOA

- ▶ Gauza bera egiteko, inplementazio desberdinak

HELBURUA

- ▶ Kolekzio bat sekuentzialki zeharkatzeko modu bat lortu, baina, bere adierazpenaren modu independentean

PATROIAK: ITERATOR

IMPLEMENTAZIOA

Arrayetan oinarritutako zerrenda	Zerrenda estekatuetan oinarritutako zerrenda
<pre>ArrayList<String> lista = new ArrayList<String>(); ... String cadena = null; Iterator<String> iter = lista.iterator(); while (iter.hasNext()){ cadena = iter.next(); ... } ...</pre>	<pre>LinkedList<String> lista = new LinkedList<String>(); ... String cadena = null; Iterator<String> iter = lista.iterator(); while (iter.hasNext()){ cadena = iter.next(); ... } ...</pre>

Ez da kolekzio konkretuaren egitura ezagutu behar osagaiak zeharkatu ahal izateko

PATROIAK: SINGLETON

MOTIBAZIOA

- ▶ Elementu bakarra puntu desberdinetik erreferentziatu behar denean, elementu hori beti berdina dela ziurtatu behar da. Instantzia bakarra egon behar da, beste guztiek objektu bera ikusteko.

PATROIAK: SINGLETON

DESKRIBAPENA

- ▶ Klaseak berak sortu bere instantzia bakarra.
- ▶ Sarrera globala klase-metodo batekin (estatikoa)
- ▶ Klaseko konstruktorea pribatua, instantzia berriak egitea ezinezkoa izateko.

PATROIAK: SINGLETON

IMPLEMENTAZIOA

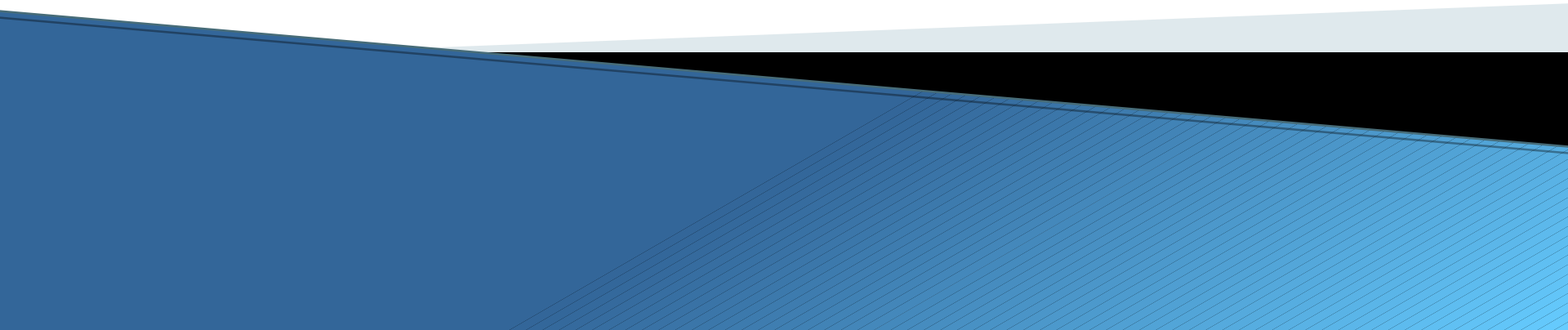
```
private static Singleton instantziaBakarra = null;

// Konstruktore pribatua, beste klaseek instantzia berriak sortu ezin
private Singleton() {}

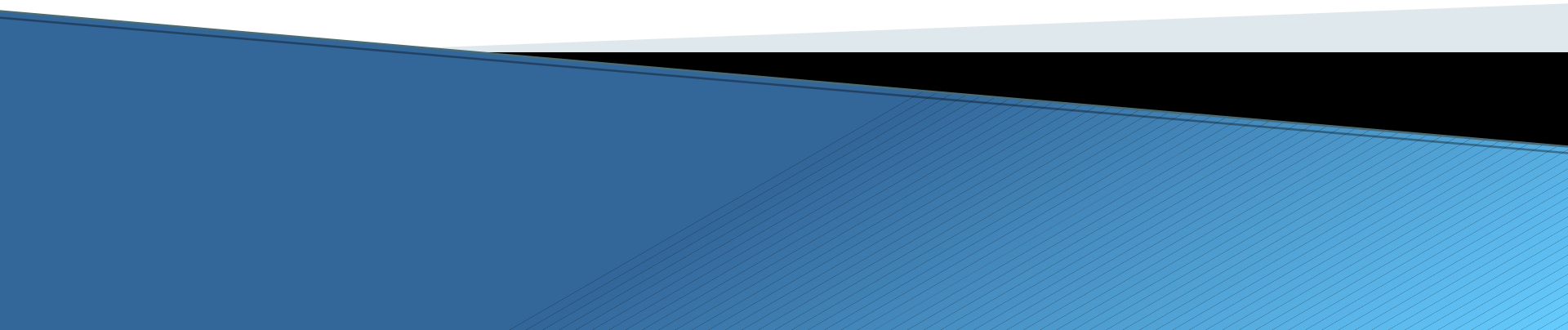
//Instantzia bakarra itzuli. Sortuta ez badago, sortu egiten du
public static Singleton getInstance() {
    if (instantziaBakarra == null) {
        instantziaBakarra = new Singleton();
    }
    return instantziaBakarra;
}
```


Diseinu Patroiak

SOFTWARE INGENIARITZA

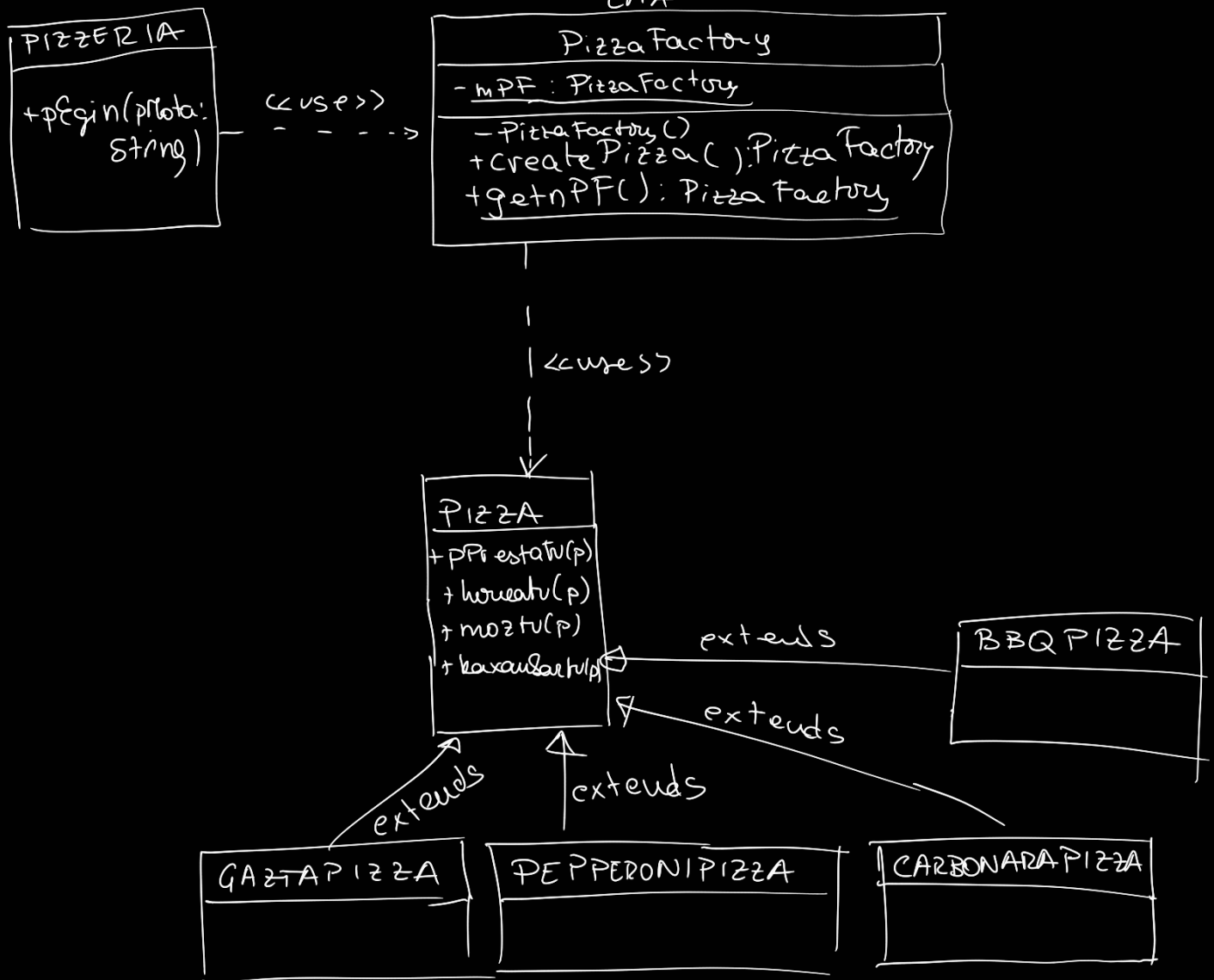


Sortzaileak



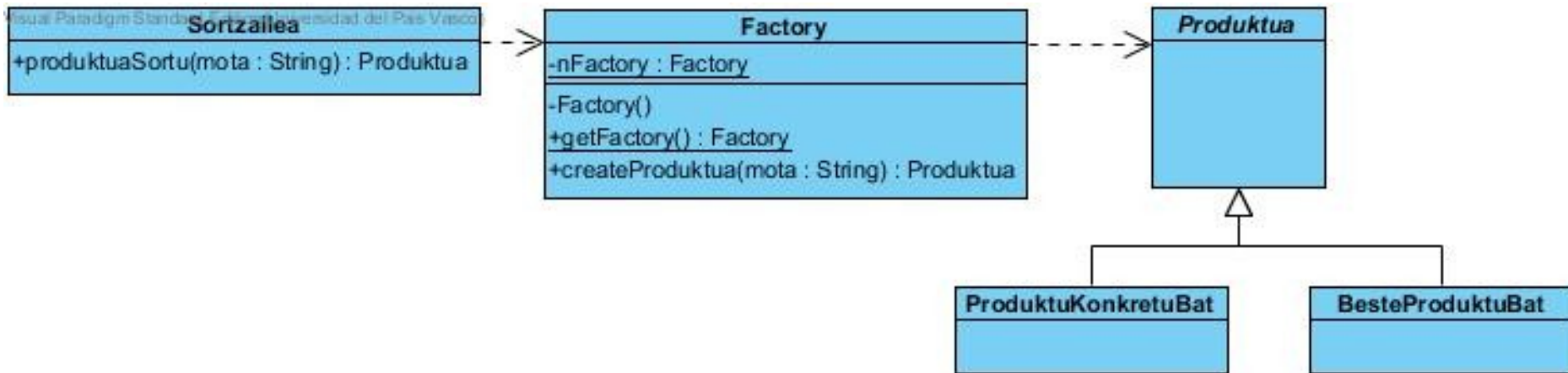
Simple Factory

The bottom of the slide features a decorative graphic. It consists of a solid dark blue shape on the left, a black horizontal band in the middle, and a light blue area on the right. The bottom-most section is a large blue area with a diagonal line pattern.



Eskema Orokorra


Factory: objektuak sortzeko interfazea definitu, baina, azpiklaseen esku klaseen instanziazioaren kudeaketa



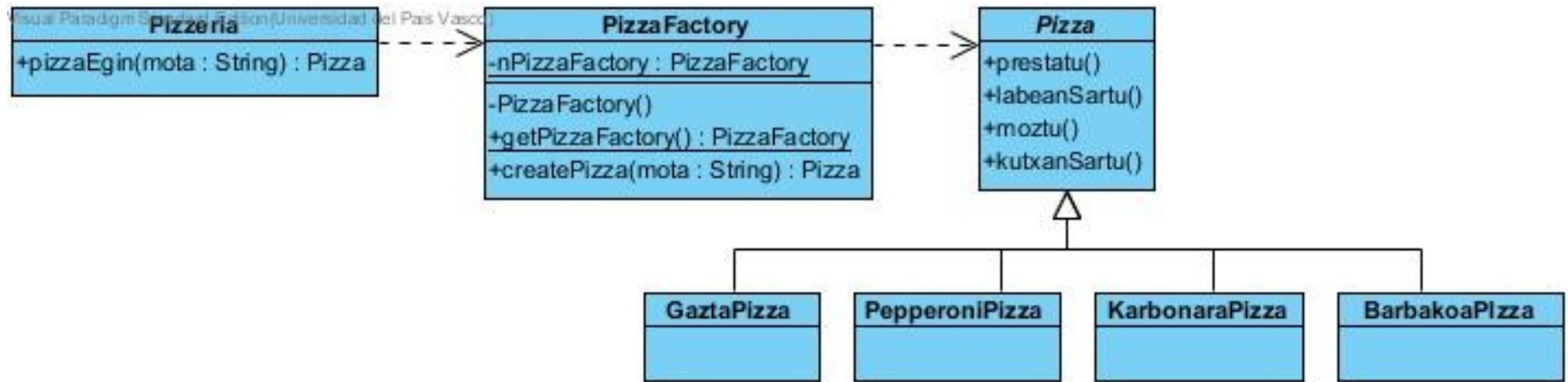
Ezaugarriak

- ▶ Objektuen sorrera faktorian kapsulatuta
- ▶ Objektuen sorrera (faktoria) eta objektuekin lan egitea (pizzeria) banatuta
- ▶ Pizza mota berri bat sortzeko
 - Klase abstraktua hedatzeko klasea sortu
 - Faktorian bi lerro gehitu
- ▶ Objektuen sorrera kontrolatu
- ▶ Mantenketa eta hedatzea erraztu

Arazoa

- ▶ Pizzeria batetako aplikazioan, hurrengoak saldu: *gazta, pepperoni, karbonara, barbakoa*
 - ▶ Pizza bakoitzerako: *prestatu, labean sartu, moztu* eta *kutxan sartu*.
 - ▶ Pizzak egiteko aplikazioaren diseinua egin, etorkizunean pizza mota gehiago egitea posible dela kontutan hartuz.
- 

Ebazpena



Ebazpena

```
public class Pizzeria {
    public Pizzeria(){
    }
    public Pizza pizzaEgin (String mota){
        Pizza nirePizza = PizzaFactory.getPizzaFactory().createPizza(mota);
        nirePizza.prestatu();
        nirePizza.labeanSartu();
        nirePizza.moztu();
        nirePizza.kutxanSartu();
        return nirePizza;
    }
    public static void main(String [ ] args){
        Pizzeria nirePizzeria = new Pizzeria();

        Pizza bbPizza = nirePizzeria.pizzaEgin("Barbakoa");
        System.out.println("Pizza eginda dago eta " + bbPizza.getClass().toString() + "
                                motakoa da!");
    }
}
```

Ebazpena

```
public class PizzaFactory {  
  
    private static PizzaFactory nPizzaFactory;  
  
    private PizzaFactory (){}  
  
    public static PizzaFactory getPizzaFactory(){  
        if (nPizzaFactory == null) {nPizzaFactory = new PizzaFactory();}  
        return nPizzaFactory;  
    }  
  
    public Pizza createPizza (String mota){  
        Pizza nirePizza = null;  
        if(mota == "Gazta"){nirePizza = new GaztaPizza();}  
        else if(mota == "Pepperoni"){nirePizza = new PepperoniPizza();}  
        else if (mota == "Karbonara"){nirePizza = new KarbonaraPizza();}  
        else if (mota == "Barbakoa"){nirePizza = new BarbakoaPizza();}  
        return nirePizza;  
    }  
}
```

Ebazpena

```
public abstract class Pizza {  
  
    public Pizza(){}  
    public void prestatu(){System.out.println("Pizza prestatu da.");}  
    public void labeanSartu(){System.out.println("Pizza labean sartu da.");}  
    public void moztu(){System.out.println("Pizza moztu da.");}  
    public void kutxanSartu(){System.out.println("Pizza kutxan sartu da.");}  
}  
  
public class BarbakoaPizza extends Pizza{  
    public BarbakoaPizza(){}  
}  
  
....
```

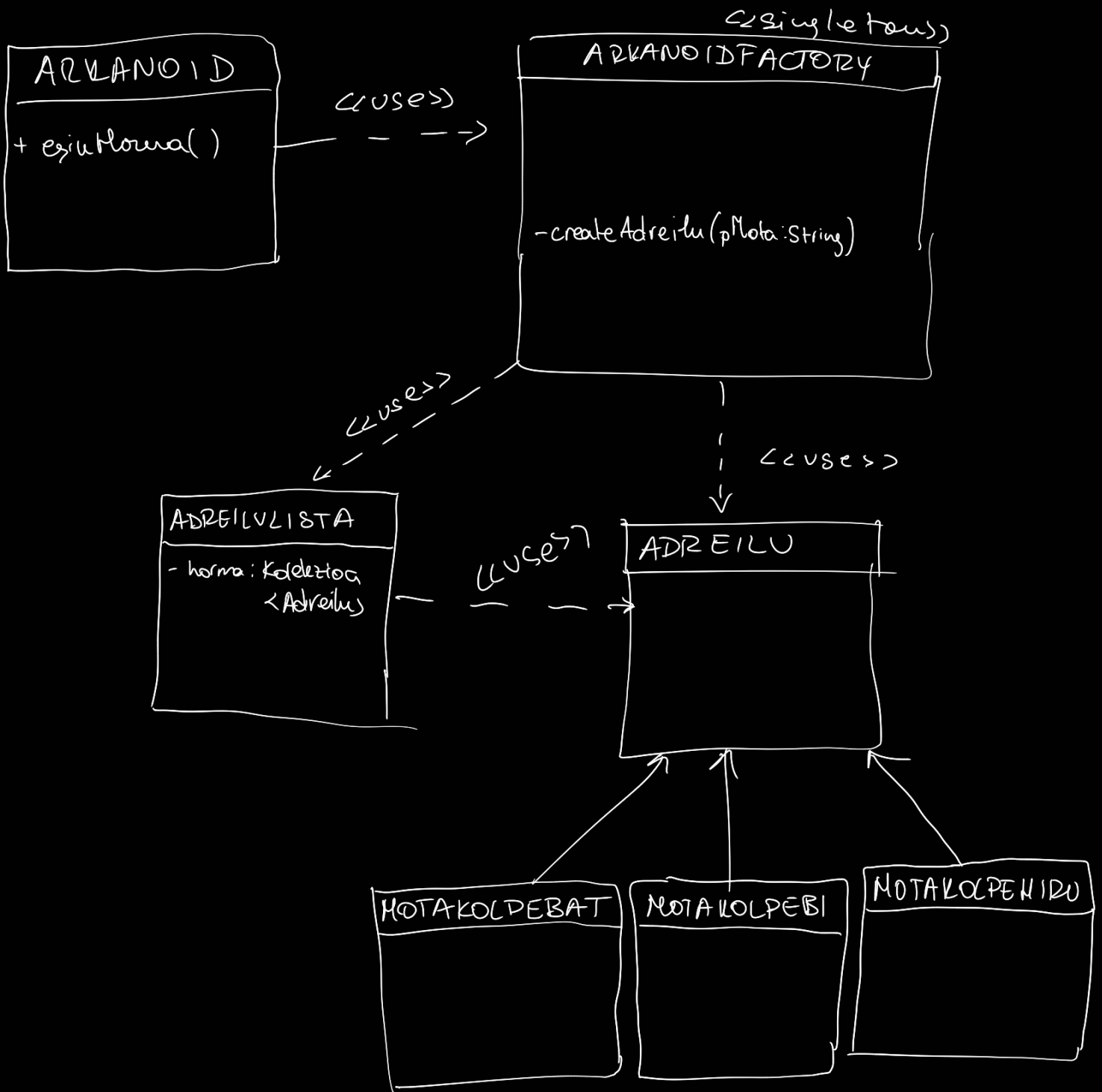
Ariketa: Arkanoid



- ▶ Arkanoid jokuan adreilu horma bat suntsitu behar da (suntsiezinak ez diren bitartean), pilota bat adreiluetan errebote eraginez.
- ▶ Adreiluak mota ezberdinekoak izan daitezke: 1, 2 edo hiru kolperen ondoren puskatzen direnak.

Ariketa: Arkanoid

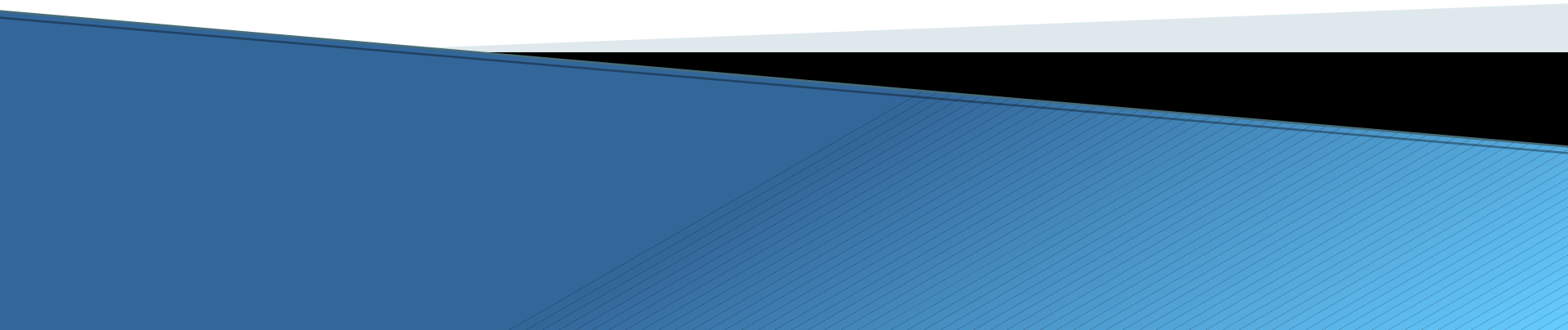
- ▶ Eskatzen da:
 - Jokoaren diseinua (Klase Diagrama)
 - Jokoaren horma sortzen duen zatiaren inplementazioa (hormaren adreiluen mota ausaz erabakitzen da).



Egiturazkoak



Facade



Zer da akoplamendua?

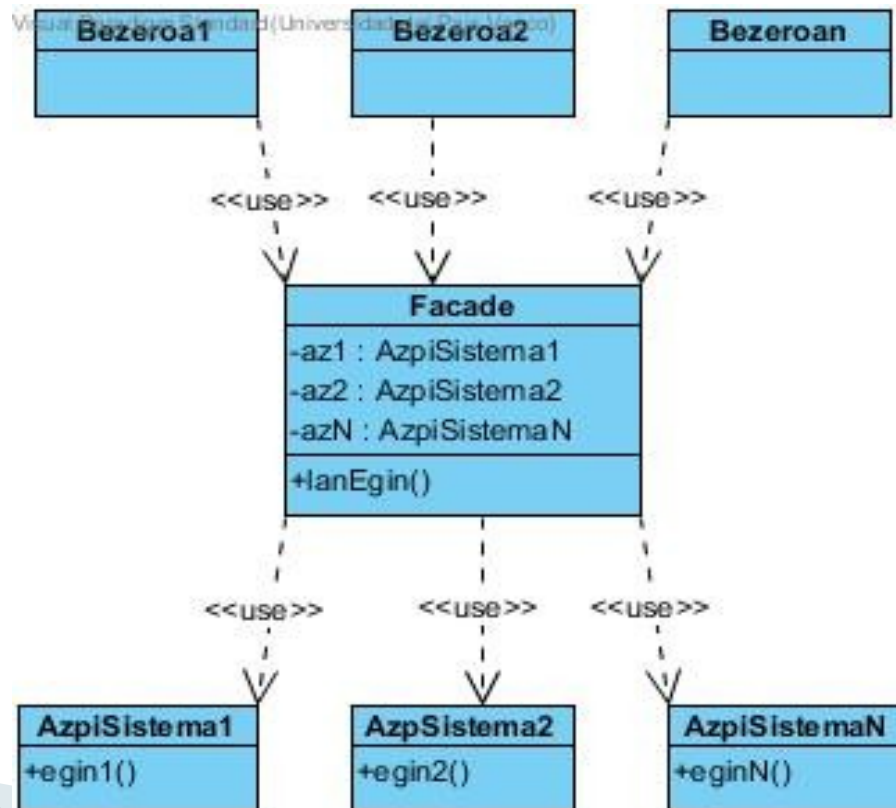
Klaseek beren artean duten **dependentzia maila** da. Zenbat eta akoplamendu txikiagoa, orduan eta eragin gutxiago izango dute sistemako aldaketek gure programan.

Ezagutza minimoaren printzipioa


- ▶ Akoplamendu ahula (Loose coupling):
 - Klase batek berarekin elkarrekintza estuan daudenak soilik ezagutu
 - Klase batek bere “lagunekin” soilik berba, ez “arrotzekin”
- ▶ Helburua: akoplamendua ahalik eta gehien murriztu

Eskema Orokorra

Facade: azpisistema multzo baten inertzazeei interfaze bateratua ezarri; hau da, bezeroari maila altuko interfazea eskaini, azpisistemak erabilterraza goak bihurtzeko



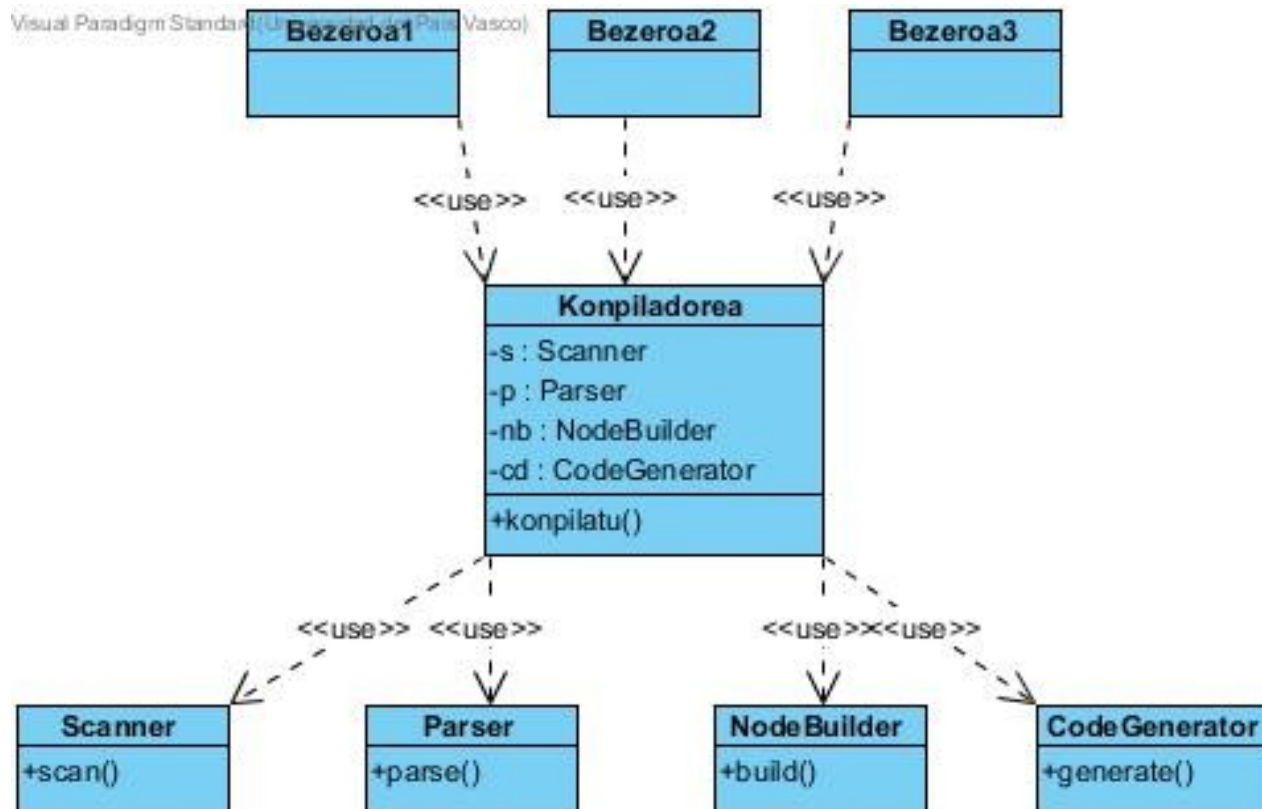
Ezaugarriak

- ▶ Bezeroa sistematik isolatu
 - ▶ Bezero/azpisistemen akoplamendu ahula
 - ▶ Azpisistemak bezeroarentzat eskuragarri, behar izanez gero
 - ▶ Sistema geruza banatu
 - ▶ Kontuan izan: bezeroek azpisistema desberdinak erabiliz gero, facade desberdinak
- 

Arazoa

- ▶ Java konpiladore baten diseinua
- ▶ Konpilatzeko, azpisistema desberdinak:
 - Scanner-ak programa irakurri
 - Parser-ak prozesatu
 - NodeBuilder-ak zuhaitza sortu
 - CodeGenerator-ak bytecode-a sortu
- ▶ Konpiladorearen klase diagrama egin, etorkizunean azpisistemak aldatu daitezkeela kontutan hartuz.

Ebazpena



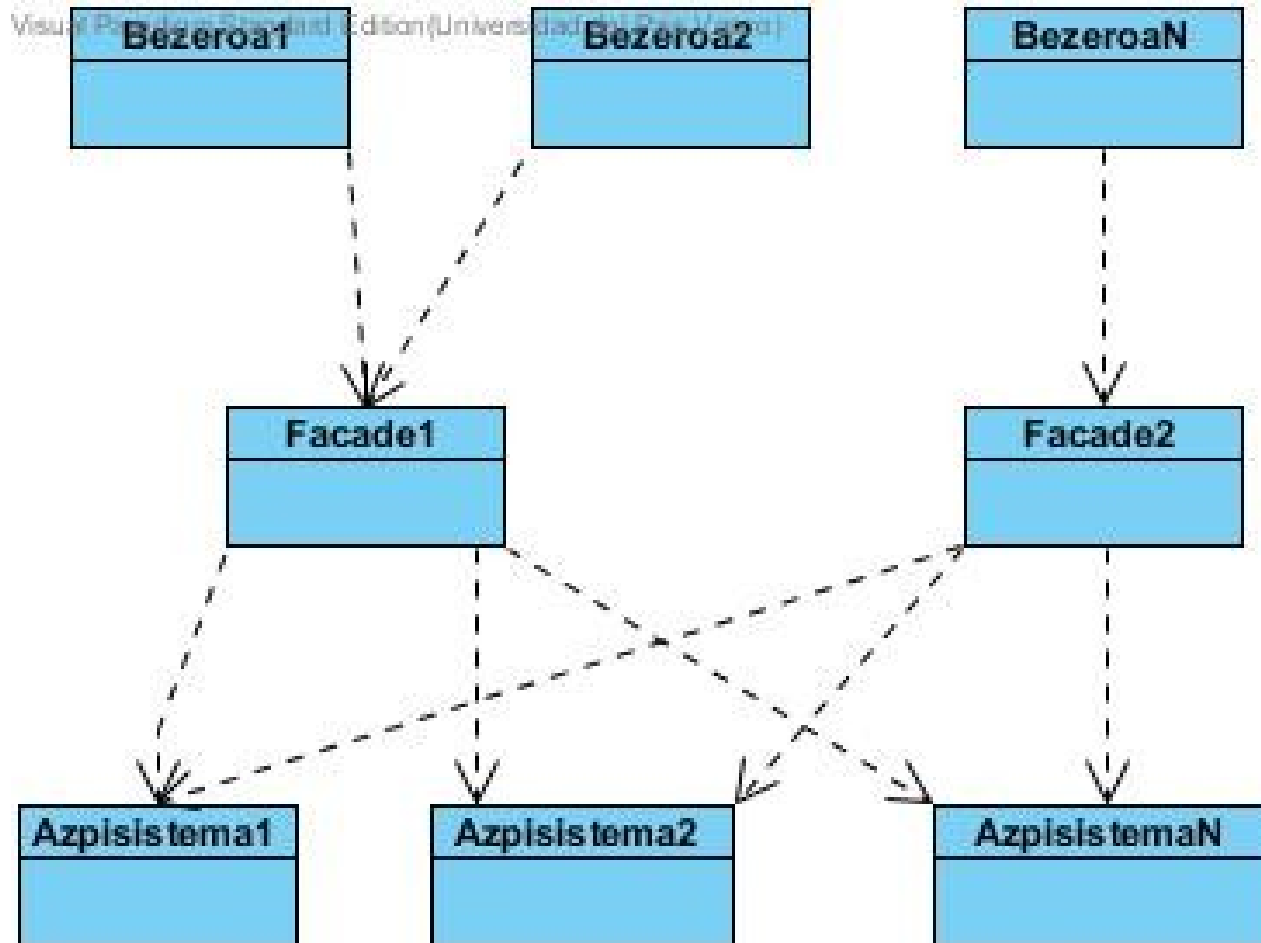
* Sinplifikatze aldera, ez dira parametroak diagraman gehitu

Ebazpena

```
public class Konpiladorea {  
    private static Konpiladorea nKonpiladorea;  
    private Scanner scanner;  
    private Parser parser;  
    private NodeBuilder nodeBuilder;  
    private CodeGenerator codeGenerator;  
  
    private Konpiladorea () {  
        scanner = new Scanner();  
        parser = new Parser();  
        nodeBuilder = new NodeBuilder();  
        codeGenerator = new CodeGenerator();  
    }  
  
    public static Konpiladorea getKonpiladorea(){...}  
  
    public void compile(){  
        scanner.scan();  
        parser.parse();  
        nodeBuilder.build();  
        codeGenerator.generate();  
    }  
}
```

Azpisistemak
isolatzen ditu

Orokortuz



Ariketa: Multimedia Gela

- ▶ Multimedia gela bat kudeatzeko sistema inplementatu.
- ▶ Gelan bi motatako ekitaldiak: *pelikula emanaldiak* eta *hitzaldiak*.
- ▶ Bi kasuetan, *pantaila jaitsi* eta *proiektorea piztu*
- ▶ Pelikula emanaldietan, gainera: *proiektorea DVD moduan jarri, DVD-a piztu, bozgorailuak piztu, bere bolumena finkatu, diskoa sartu eta diskoa martxan jarri.*

Ariketa: Multimedia Gela

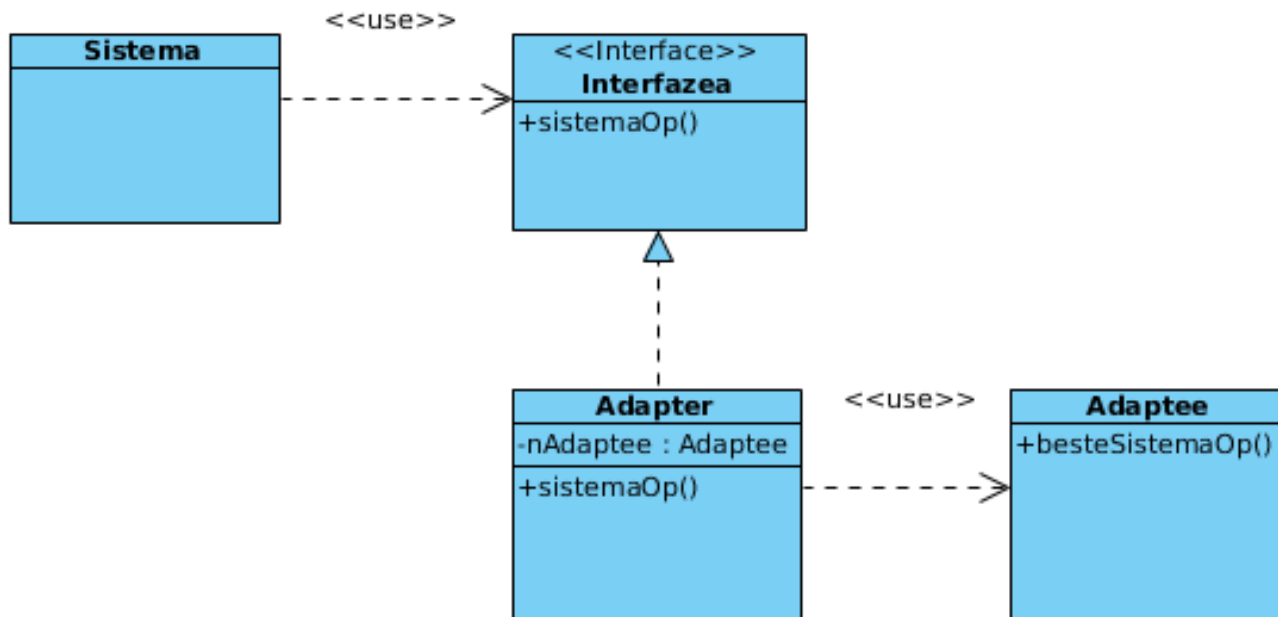
- ▶ Hitzaldietan, gainera: *proiektorea PC moduan jarri, ordenagailua piztu eta aurkezpena martxan jarri*
- ▶ Sistemaren klase diagrama eta ekitaldi mota bakoitza kudeatzeko zatiaren inplementazioa

Adapter



Eskema Orokorra

Adapter: interfaze bateraezinei elkarrekin lan egiteko aukera eman. Gure sistemako funtzionalitate baliokidea dauka bezzeroaren interfazeak, baina intefaze desberdina; bitartekari lana egiten du.



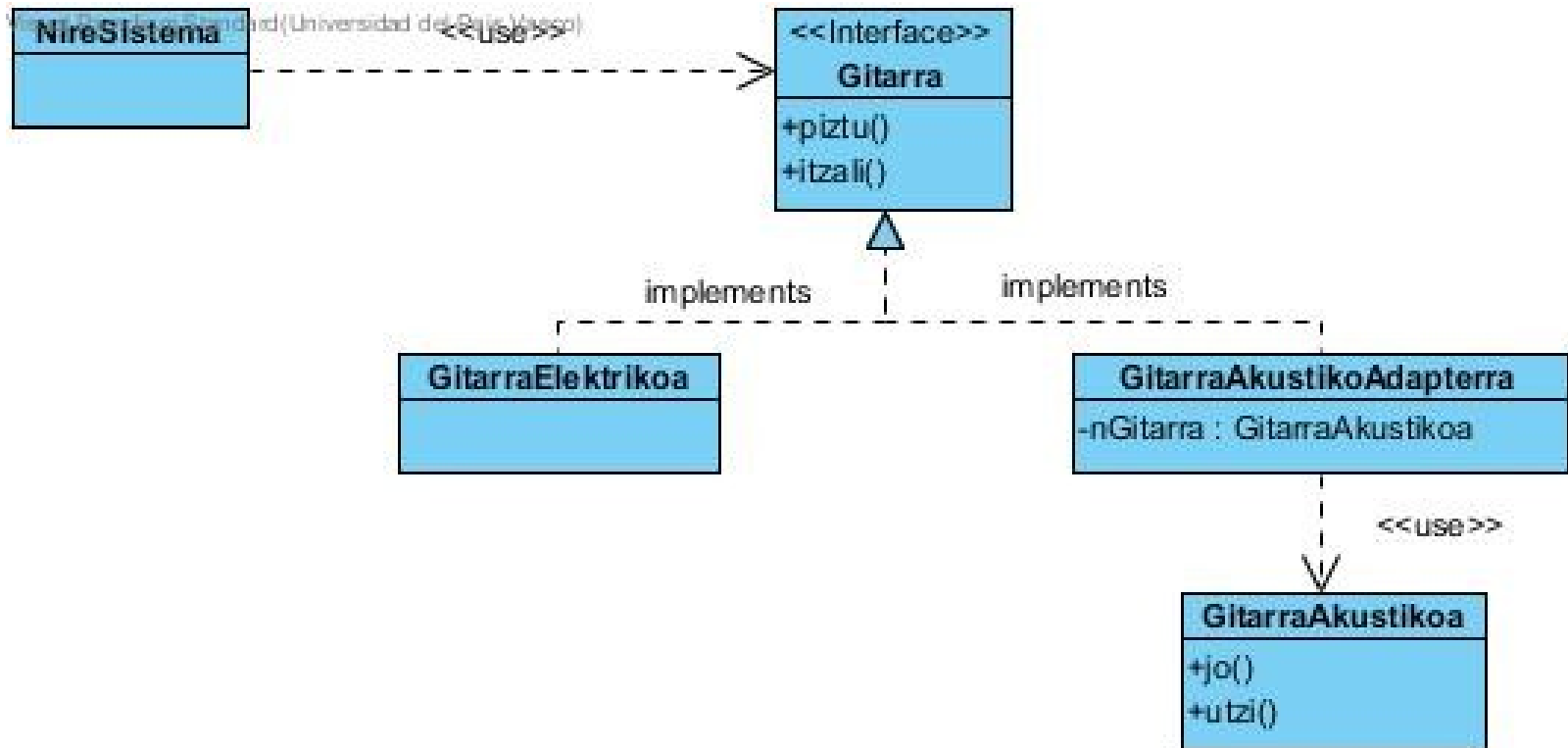
Ezaugarriak

- ▶ Berrerabilgarritasuna hobetu
- ▶ Hedapena erraztu
- ▶ Adaptee ez da ukitzen
 - Kodea agian ez dago eskuragarri

Arazoa

- ▶ Musika tresnak simulatzeko sistema dugu.
 - Gitarra simulatzeko “Gitarra” interfazea dugu, *piztu* eta *itzali* metodoekin.
 - Interfaze hori “GitarraElektriko” klaseak inplementatzen du.
- ▶ Beste sistema bateko “GitarraAkustiko” klasea berrerabiliko dugu, baina, *jo* eta *utzi* metodoak ditu.
- ▶ Sistemaren klase diagrama egin, berrerabilgarria izan behar duela kontutan hartuz.

Ebazpena



Ebazpena

```
public interface Gitarra {  
    public void piztu();  
    public void itzali();  
}
```

Klase abstraktu bat
ere izan daiteke

```
public class GitarraElektrikoa implements Gitarra {  
    public void piztu(){...}  
    public void itzali () {...}  
}
```

```
public class GitarraAkustikoa {  
    public void jo(){...}  
    public void utzi(){...}  
}
```

Ez du
GitarraAkustikoa
aldatzen

```
public class GitarraAkustikoAdapterra implements Gitarra {  
    private GitarraAkustikoa gitarraAkustikoa = new GitarraAkustikoa();  
  
    public void piztu(){gitarraAkustikoa.jo();}  
    public void itzali(){gitarraAkustikoa.utzi();}  
}
```


Ariketa: Motorrak

- ▶ Audi kotxeak kontrolatzeko sistema dugu
 - Motorrak hiru operazio: *piztu*, *azeleratu* eta *itzali*.
- ▶ Motore elektrikoak kudeatzeko sistema beste enpresa bati erosi diogu. Kasu horretan, motoreek *konektatu*, *aktibatu*, *azkartu*, *gelditu* eta *deskonektatu* operazioak dituzte.

Ariketa: Motorrak

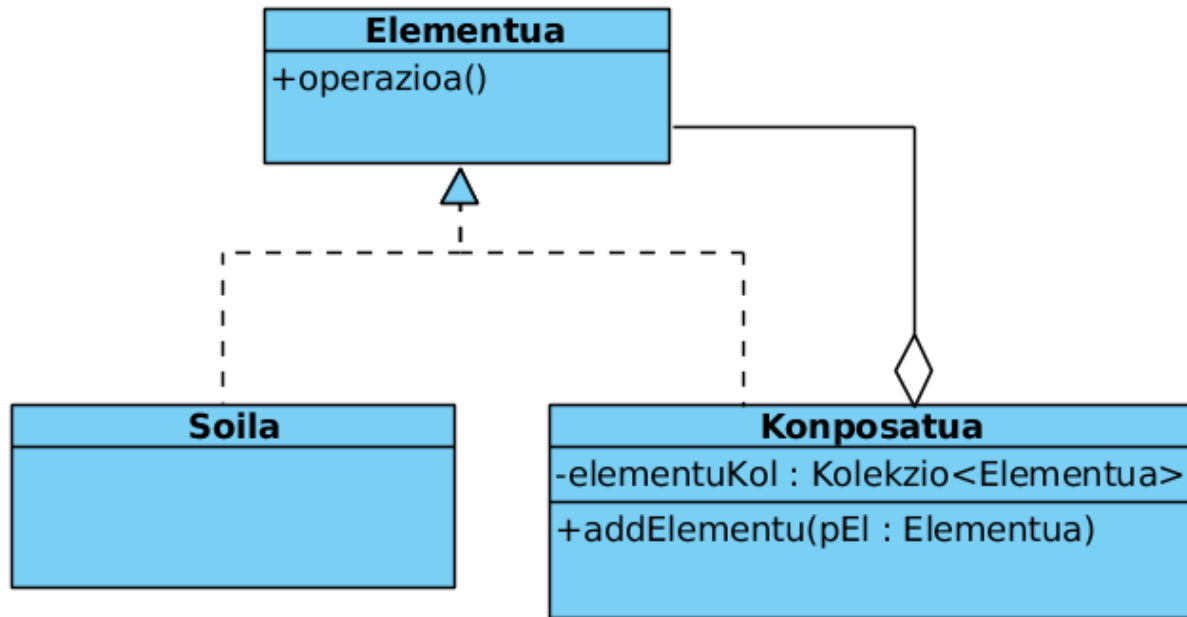
- ▶ Hurrengoa eskatzen da:
 - Sistemaren diseinua (Klase Diagrama)
 - Gure sisteman motor elektrikoak sartzea ahalbidetuko duen zatiaren inplementazioa.

Composite

The bottom of the slide features a decorative graphic consisting of several overlapping geometric shapes. On the left, there is a solid dark blue trapezoidal shape. To its right, a black horizontal band is visible. Further right, a light blue trapezoidal shape overlaps the black band. The bottom right corner is filled with a series of parallel, diagonal light blue lines, creating a textured effect.

Eskema Orokorra

Composite: aplikazioan “banakako” objektuak eta “konposatuak” modu berean erabiltzea ahalbidetzen du. Zuhaitz motako hierakietan txertatzen ditu objektu horiek.



Ondorioak

- ▶ Banakako elementuak (hostoak) eta konposatuak (nodoak) zuhaitz egitura berean txertatu
 - “Part-whole” hierarkiak
 - Konposatuek elementu soilez zein konposatuez esatuta egon daitezke
- ▶ Objektu guztiek interfaze bera
 - Nodo zein hosto, era berean tratatu

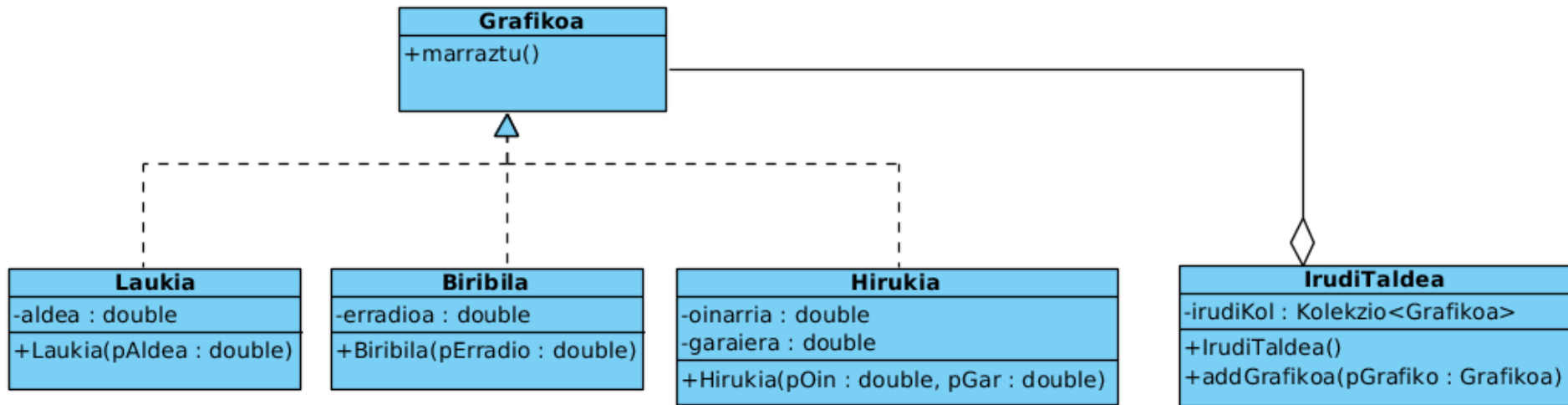
Arazoa

- ▶ Aplikazio baten elementu grafikoen informazioa biltzeko klaseak behar ditugu. Adibidez, *biribilak*, *laukiak* eta *hirukiak*.
- ▶ *Irudi multzoak* tratatu behar ditugu. Programak zenbait irudi batera lantzeko aukera egin behar du, objektu bakarra balitz bezala; pantailan zehar mugitzeko, koloreztatzeko edo berdimentsionatzeko.

Arazoa

- ▶ Aplikazioa diseinatzeko, figura mota bakoitzarentzat (*irudi* zein *irudi multzo*) klase bat definitu daiteke, dagokion *marraztu()* metodoarekin.
- ▶ Baina, nola definitu irudi multzo bat irudi bakarra balitz bezala kudeatzeko?

Ebazpena



Irudi talde batek Grafiko kolekzio bat du. Grafiko horiek *Biribilak*, *Laukiak* edo beste irudi talde bat izan daitezke

Ebazpena

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

```
public interface Grafikoa{  
    public void marraztu();  
}
```

```
// ELEMENTU KONPOSATUA (NODOA)
```

```
class IrudiTaldea implements Grafikoa{
```

```
    private List<Grafikoa> irudiKol = new ArrayList<Grafikoa>();
```

```
    public void marraztu() {  
        for (Grafikoa grafikoa : irudiKol) {grafikoa.marraztu();}  
    }
```

```
    public void addGrafiko(Grafikoa grafikoa) {  
        irudiKol.addGrafikoa(grafikoa);  
    }  
}
```

Grafiko kolekzioa,
hosto zein nodo

Grafiko guztiak marraztu,
hosto zein nodo

Grafikoa gehitu, hosto
zein nodo

Ebazpena

// ELEMENTU SOILA (HOSTOA)

```
class Hirukia implements Grafikoa {  
    public void marraztu() {  
        System.out.println("Hirukia");  
    }  
}
```

Ebazpena

```
/** Bezeroa*/  
public class Proba {  
  
    public static void main(String[] args) {  
        //HOSTOAK hasieratu  
        Hirukia hirukia1 = new Hirukia();  
        Hirukia hirukia2 = new Hirukia();  
        Hirukia hirukia3 = new Hirukia();  
        Hirukia hirukia4 = new Hirukia()  
        //NODOAK hasieratu  
        IrudiTaldea talde1 = new IrudiTaldea();  
        IrudiTaldea talde2 = new IrudiTaldea ();  
        IrudiTaldea talde3 = new IrudiTaldea ();  
    }  
}
```

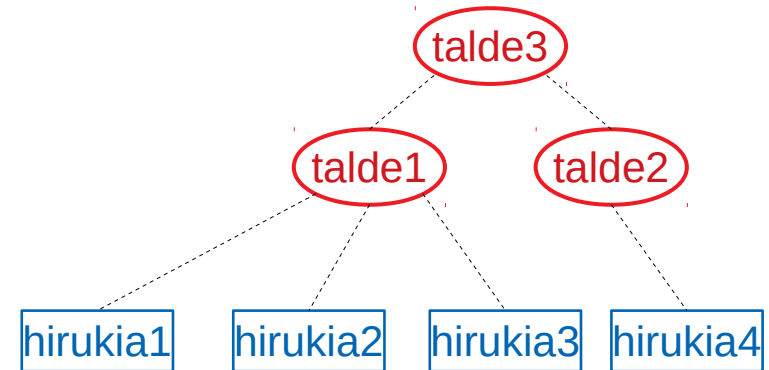
Ebazpena

//NODOAK osatu

```
talde1.add(hirukia1);  
talde1.add(hirukia2);  
talde1.add(hirukia3);  
talde2.add(hirukia4);  
talde3.add(talde1);  
talde3.add(talde2);
```

1. nodoa
2. nodoa
3. nodoa

```
talde3.marraztu();  
}  
}
```



Errekurtsiboki, "Hiruki"
guztiak marraztu

Ariketa

- ▶ Laborategian ikusitako Swing liburutegiko osagai eta edukiontzien egitura bat sortuko dugu. Demagun hurrengoak ditugula soilik:
 - Konposatuak: *JFrame* eta *JPanel*
 - Soilak: *JButton* eta *JLabel*

Erreferentziak

► Informazio gehiago:

- Gamma, E. et al. *Designs Patterns, Elements of Reusable Object Oriented Software*. Addison Wesley.
- Patterns Home Page: <http://hillside.net/patterns/>
- Liburuak patroiei buruz:
<https://cutt.ly/vrTamMP>
<http://hillside.net/patterns/books/>
<http://www.javacamp.org/designPattern/>
<http://www.dofactory.com/net/design-patterns>