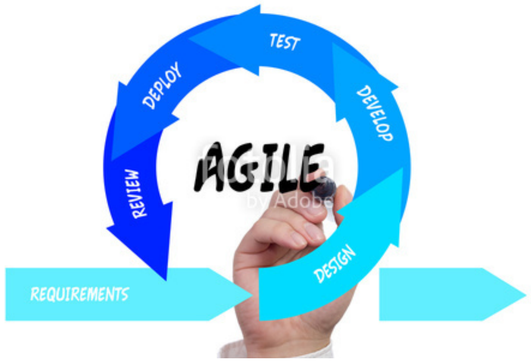


Classe Problemes Setmana 10: Disseny: Intro Patrons de Disseny

Anna Puig

Enginyeria Informàtica
Facultat de Matemàtiques i Informàtica,
Universitat de Barcelona
Curs 2021/22

UNIVERSITAT DE BARCELONA

Temari

1. Introducció al procés de desenvolupament del software
2. Anàlisi de requisits i especificació
3. Disseny
   - 3.1 Introducció
   - 3.2 Patrons arquitectònics
   - 3.3 Criteris de Disseny: G.R.A.S.P.
   - 3.4 Principis de Disseny: S.O.L.I.D.
   - 3.5 **Patrons de Disseny**
4. Del disseny a la implementació
5. Ús de frameworks de testing

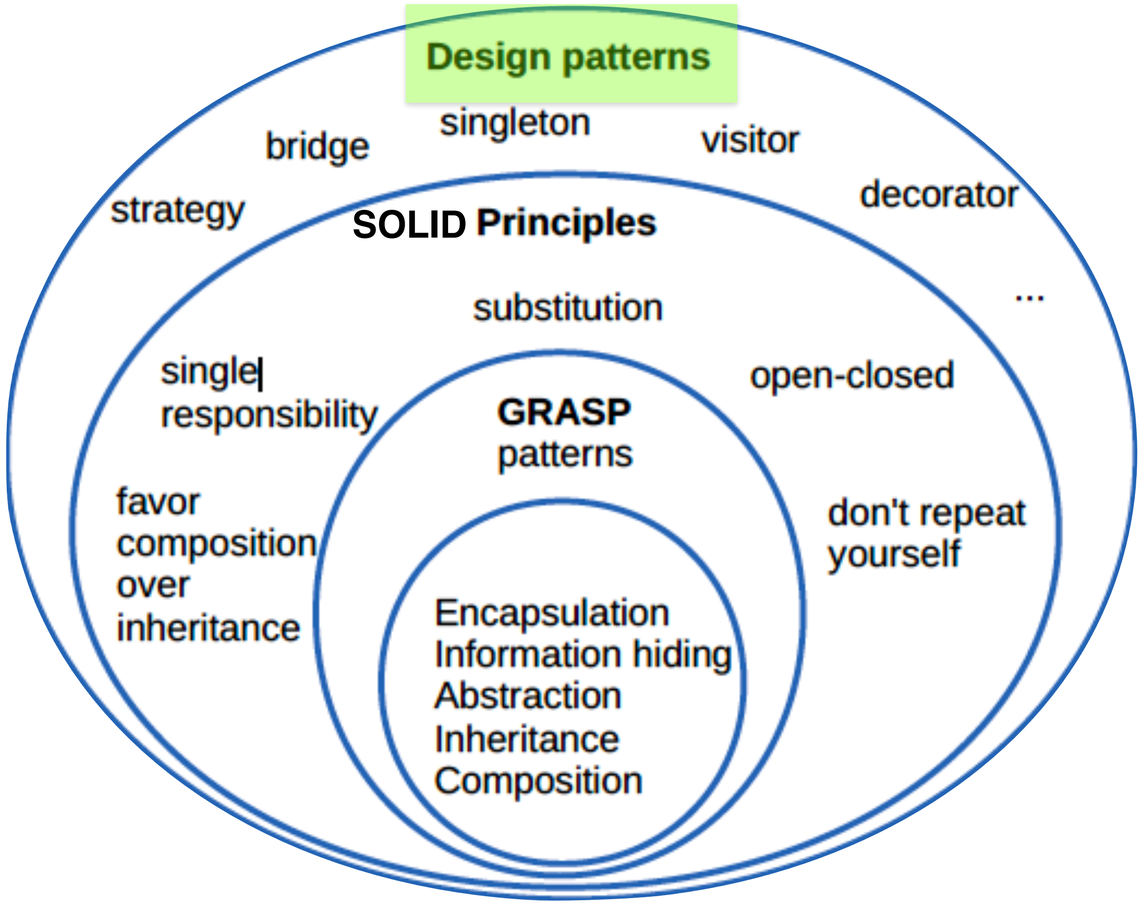
Sessió 15.11.2021

Activitats

1. Repas de Singleton
2. Patró Strategy
3. Com aplicar **patrons als problemes**?
    - exemple projecte MovieTheater
    - exemple projecte dels Ànecs

3.5. Patrons de disseny

No hi ha una metodologia que doni el *millor disseny* però hi han principis, heurístiques i patrons que hi poden ajudar



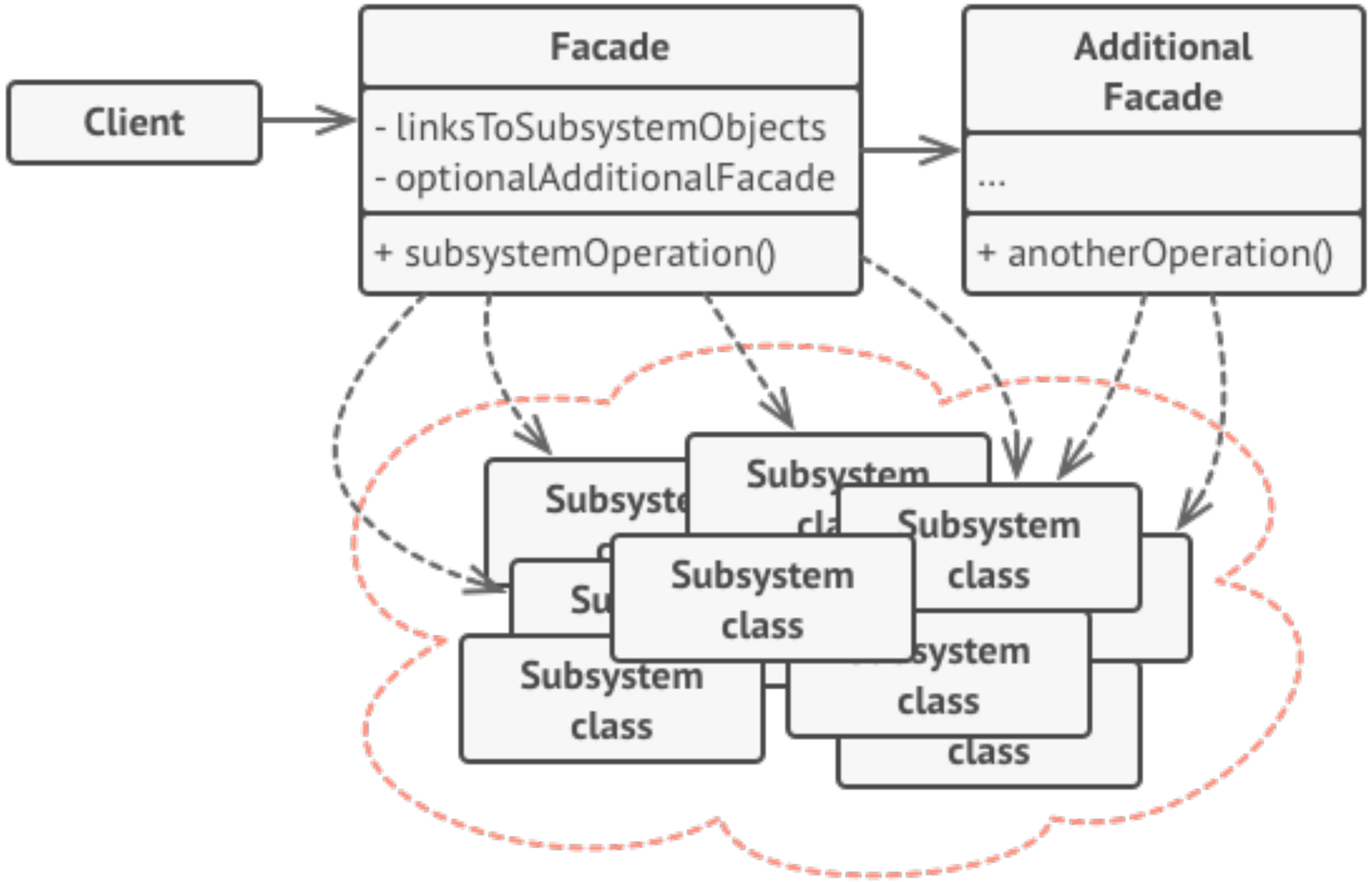
3.5. Patrons de disseny

Propòsit → **Àmbit ↓**	**CREACIÓ**	**ESTRUCTURA**	**COMPORTAMENT**
CLASSE	• Factory method	• class Adapter	• Interpreter • Template method
OBJECTE	• Abstract Factory • Builder • Prototype • Singleton • Object pool	• Object Adapter • Bridge • Composite • Decorator • **Facade** • Flyweight • Proxy	• Chain of Responsability • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

Patró Façana (Facade)

- **Facade** – Proporciona una interfície unificada a un conjunt d'utilitats d'un subsistema complex. Defineix una interfície a alt nivell que fa que el subsistema sigui més facil d'usar.

3.5.1. Patrons de disseny: Patró Façana (Facade)



Patró Singleton

- **Singleton** – Assegura una classe que només té una única instància en tota l'aplicació i proporciona l'accés global a aquesta classe.

Patró Singleton: Exemple

```
public class Singleton {


   private static Singleton uniqueInstance = new Singleton();

   private Singleton() {}

   public static Singleton getInstance() {
      return uniqueInstance;
   }

   // other useful methods here
  public String getDescription() {
      return description+" de dins del mètode";
   }

}
```

Aquí hi ha **eager** initialization!

Patró Singleton: Exemple

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

         System.out.println("Comença el main");

         System.out.println(Singleton.description);

        Singleton singleton = Singleton.getInstance();

        System.out.println(singleton.getDescription());
   }
}
```

Quan i com es reserva la memòria de Singleton?

Patró Singleton: Exemple

```
public class Singleton {


   public static String description = "I'm a statically initialized Singleton!”;

   private static Singleton uniqueInstance = new Singleton();

   private Singleton() {}

   public static Singleton getInstance() {
      return uniqueInstance;
   }

   // other useful methods here
   public String getDescription() {
      return description+" de dins del mètode";
   }
}
```

Patró Singleton: Exemple

Aquí hi ha **eager** initialization!

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

         System.out.println("Comença el main");

         System.out.println(Singleton.description);

        Singleton singleton = Singleton.getInstance();

        System.out.println(singleton.getDescription());
   }
}
```

Output:

```
Comença el main
I'm a statically initialized Singleton!
I'm a statically initialized Singleton! de dins del mètode
```

Patró Singleton: Patró clàssic

```
public class Singleton {
   private static Singleton uniqueInstance;

   private Singleton() {}

   public static Singleton getInstance() {
      if (uniqueInstance == null) {
         uniqueInstance = new Singleton();
      }
      return uniqueInstance;
   }

   // other useful methods here
   public String getDescription() {
      return "I'm a classic Singleton!";
   }
}
```

Aquí hi ha lazy initialization!

Patró Singleton: millor opció en Java

```
public enum Singleton {
   INSTANCE;


 // other useful atributes here

 // other useful methods here
  public String getDescription() {
      return description+" de dins del mètode";
   }
}
```

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

        Singleton singleton = Singleton.INSTANCE;

        System.out.println(singleton.getDescription());
   }
}
```

Eager initialization, thread-safe, serialitzable

Patró Singleton (Projecte)

Projecte del campus: singleton

previ :

- VariableGlobal (main) i SenseSingleton.java: No hi ha singleton
- SingletonClient (main) i Singleton.java: No hi ha singleton

stat:

- atribut estàtic dins de la classe Singleton: eager Singleton

classic:

- atribut estàtic dins de la classe Singleton amb allocatació a la primera crida de getInstance: lazy Singleton

threadsafe:

- codi protegit per a la sincronitació de múltiples threads

subclases:

- codi exemple de com derivar Singletons

Chocolote:

- singleton via Enum

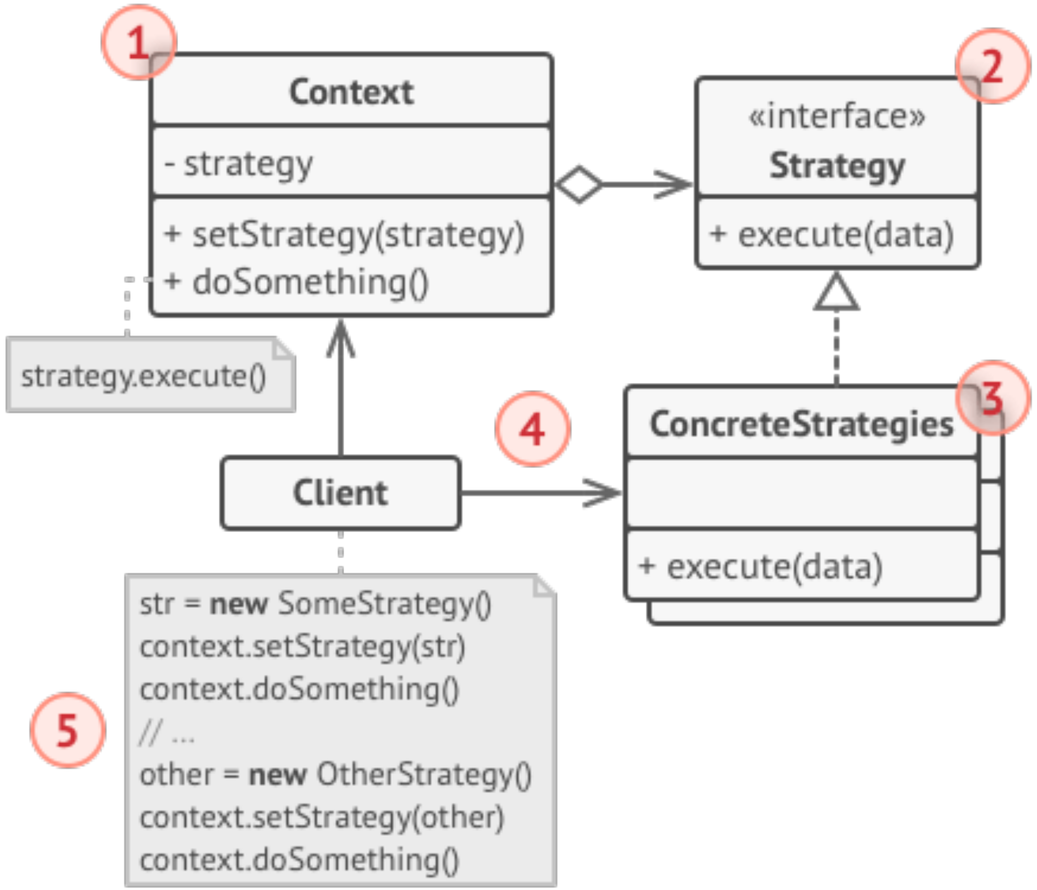
Patró Strategy

- **Strategy** – Defineix una família d'algorismes, els encapsula un a un i els fa intercanviables. Permet que l'algorisme canvïi independentment del client que l'usa.

Passos per solucionar els problemes de patrons: Projecte HomeCinema

1. **Problema** identificat a solucionar (principis que es vulneren…)
2. **Identificació del** patró a aplicar amb la seva solució genèrica
3. **Aplicació** del patró al problema
4. Programa principal o client que **usa** el patró
5. **Anàlisi** del patró utilitzat

Patró Strategy



1. El **`Context`** no coneix res de les estratègies concretes, només les sap executar
2. La classe **`Strategy`** declara la interfície comuna a tots els algorismes
3. Les implementacions concretes de les estratègies estan a les classes **`ConcreteStrategies`**
4. El **`Client`** crea l'estratègia concreta que vol fer servir
5. El **`Client`** passa al **`Context`** l'estratègia (setStrategy) i delega en el **`Context`** que l'executi.

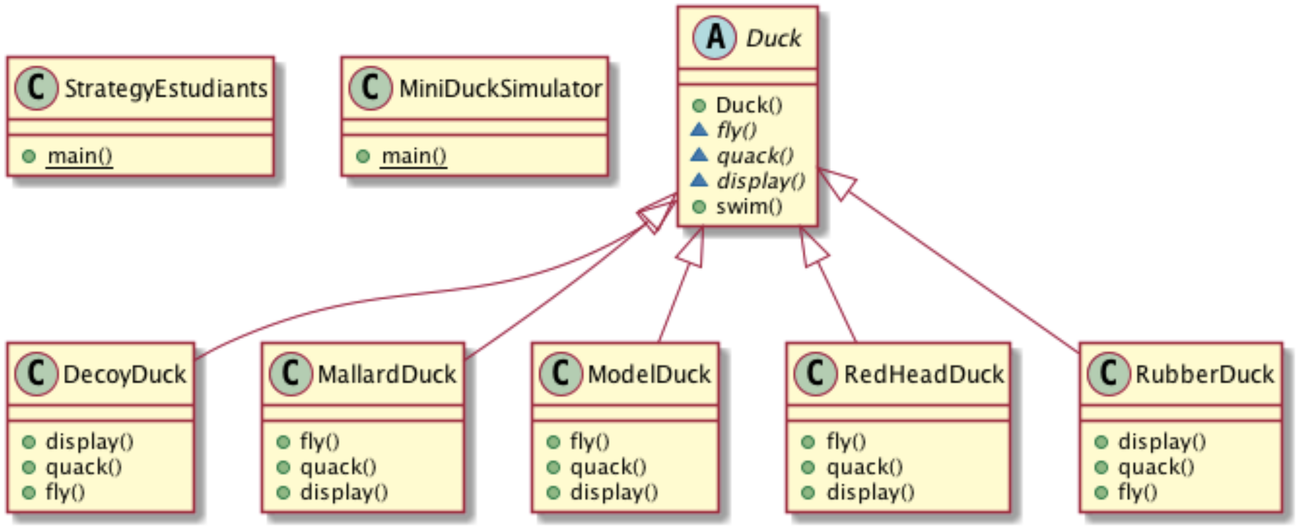
4. Passos per solucionar els problemes de patrons

Problema dels ànecs:

Es vol dissenyar una aplicació que simuli diferents tipus d'ànecs.

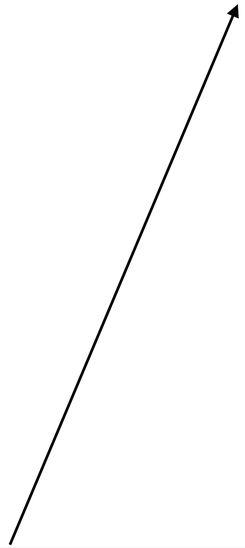
Els ànecs naden (swim), alguns parlen (quack) i alguns volen (fly). Com veureu hi ha una classe Duck, amb els mètodes quack(), swim(), fly() i display(). D'aquesta classe Duck, hereten les classes MallardDuck, ReadheadDuck, RubberDuck, DecoyDuck amb diferents implementacions dels mètodes fly() i quack().

Fixa't que quan volen, tots volen igual i, quan parlen, tots parlen igual. Identifiqueu quins principis S.O.L.I.D. vulnera. Com ho resoldríeu?



1. Problema a identificar

```java
public abstract class Duck {
        public Duck() {
        }
        abstract void fly();
        abstract void quack();
        abstract void display();

        public void swim() { System.out.println("All ducks float, even decoys!"); }
}
```

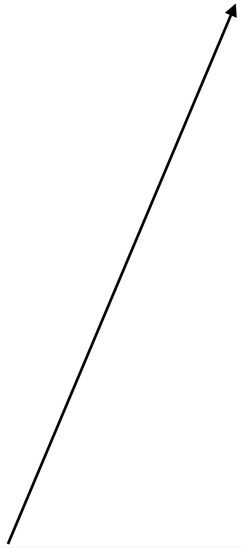



```java
public class MallardDuck extends Duck {
        public void fly() { System.out.println("I'm flying!!"); }
        public void quack() { System.out.println("Quack"); }
        public void display() { System.out.println("I'm a real Mallard duck"); }
}
```

```java
public class DecoyDuck extends Duck {
        public void display() { System.out.println("I'm a duck Decoy"); }
        public void quack() { System.out.println("<< Silence >>"); }
        public void fly() { System.out.println("I can't fly"); }
}
```

1. Problema a identificar

```java
public abstract class Duck {
        public Duck() {
        }
        abstract void fly();
        abstract void quack();
        abstract void display();

        public void swim() { System.out.println("All ducks float, even decoys!"); }
}
```




```java
public class MallardDuck extends Duck {
        public void fly() { System.out.println("I'm flying!!"); }
        public void quack() { System.out.println("Quack"); }
        public void display() { System.out.println("I'm a real Mallard duck"); }
}
```

```java
public class DecoyDuck extends Duck {
        public void display() { System.out.println("I'm a duck Decoy"); }
        public void quack() { System.out.println("<< Silence >>"); }
        public void fly() { System.out.println("I can't fly"); }
}
```

1. Problema a identificar

```java
public abstract class Duck {
        public Duck() {
        }
        abstract void fly();
        abstract void quack();
        abstract void display();

        public void swim() { System.out.println("All ducks float, even decoys!"); }
}
```

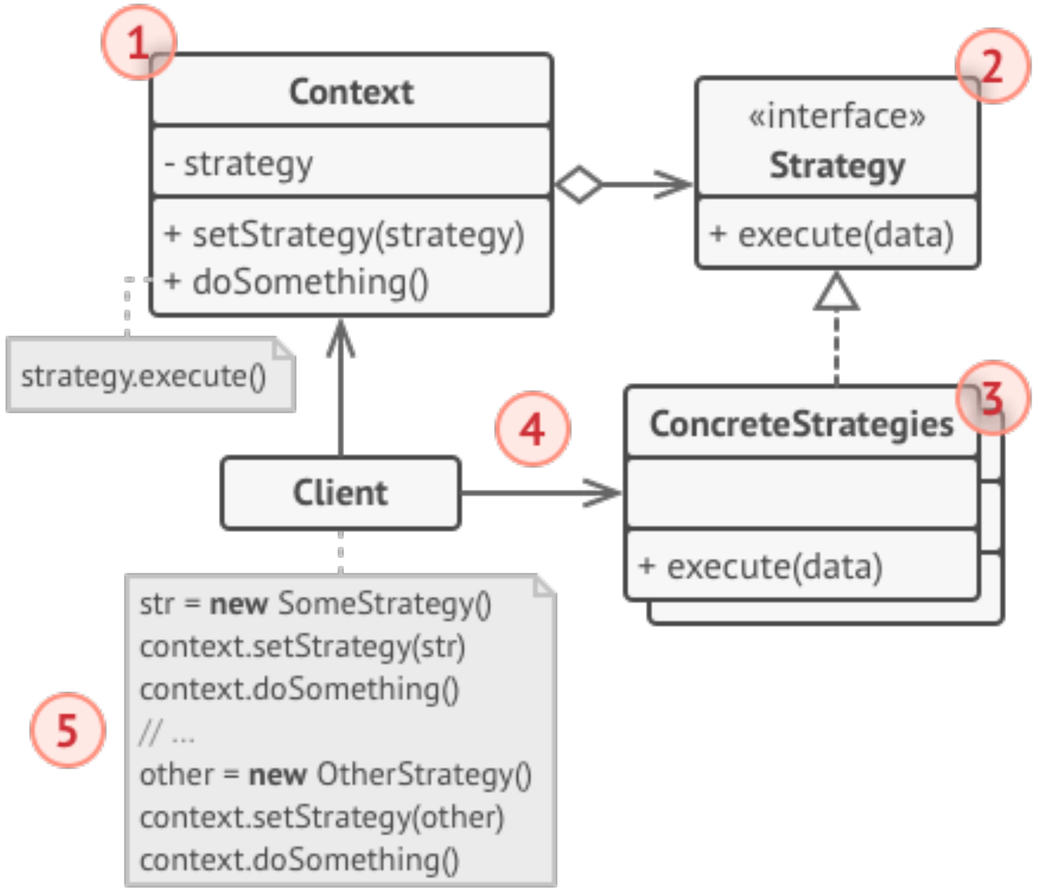
```java
public class MallardDuck extends Duck {
        public void fly() { System.out.println("I'm flying!!"); }
        public void quack() { System.out.println("Quack"); }
        public void display() { System.out.println("I'm a real Mallard duck"); }
}
```

```java
public class DecoyDuck extends Duck {
        public void display() { System.out.println("I'm a duck Decoy"); }
        public void quack() { System.out.println("<< Silence >>"); }
        public void fly() { System.out.println("I can't fly"); }
}
```

Liskov!!

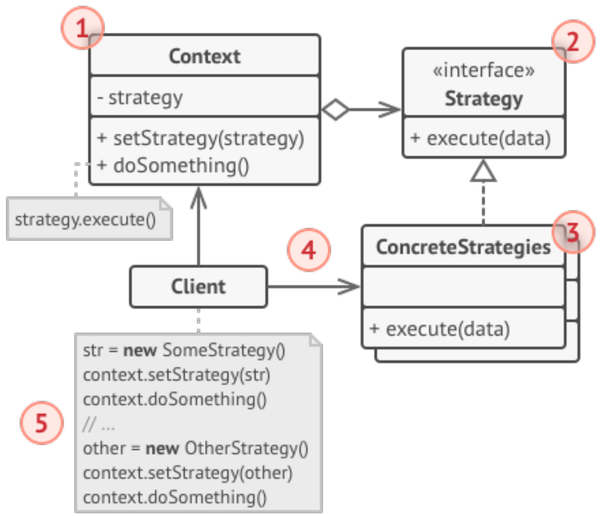
i repeticio de codi….!!

2. Patró a aplicar: Strategy



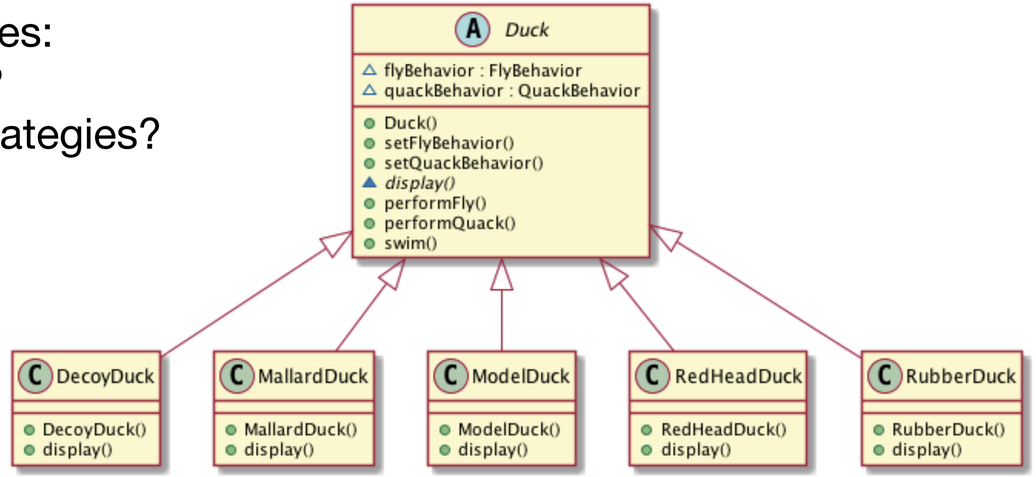
1. El **`Context`** no coneix res de les estratègies concretes, només les sap executar
2. La classe **`Strategy`** declara la interfície comuna a tots els algorismes
3. Les implementacions concretes de les estratègies estan a les classes **`ConcreteStrategies`**
4. El **`Client`** crea l'estratègia concreta que vol fer servir
5. El **`Client`** passa al **`Context`** l'estratègia (setStrategy) i delega en el **`Context`** que l'executi.

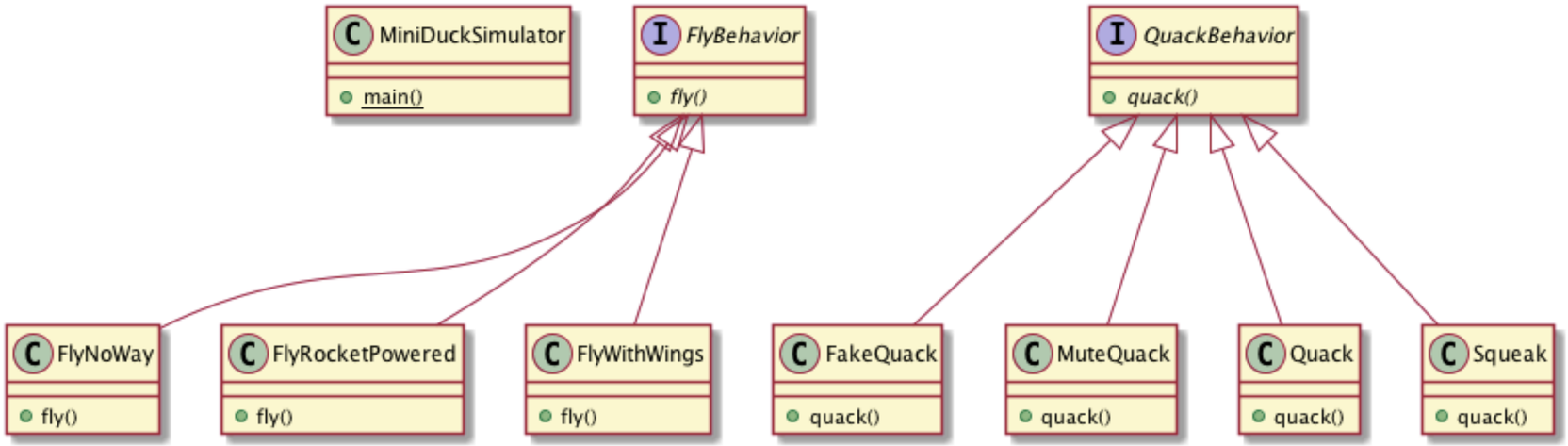
3. Aplicar el patró



Contestar les preguntes:

1. Qui és el Context?
2. Quines són les Strategies?
3. Qui és el Client?





4. Com funciona el main?

Abans d'aplicar el patró:

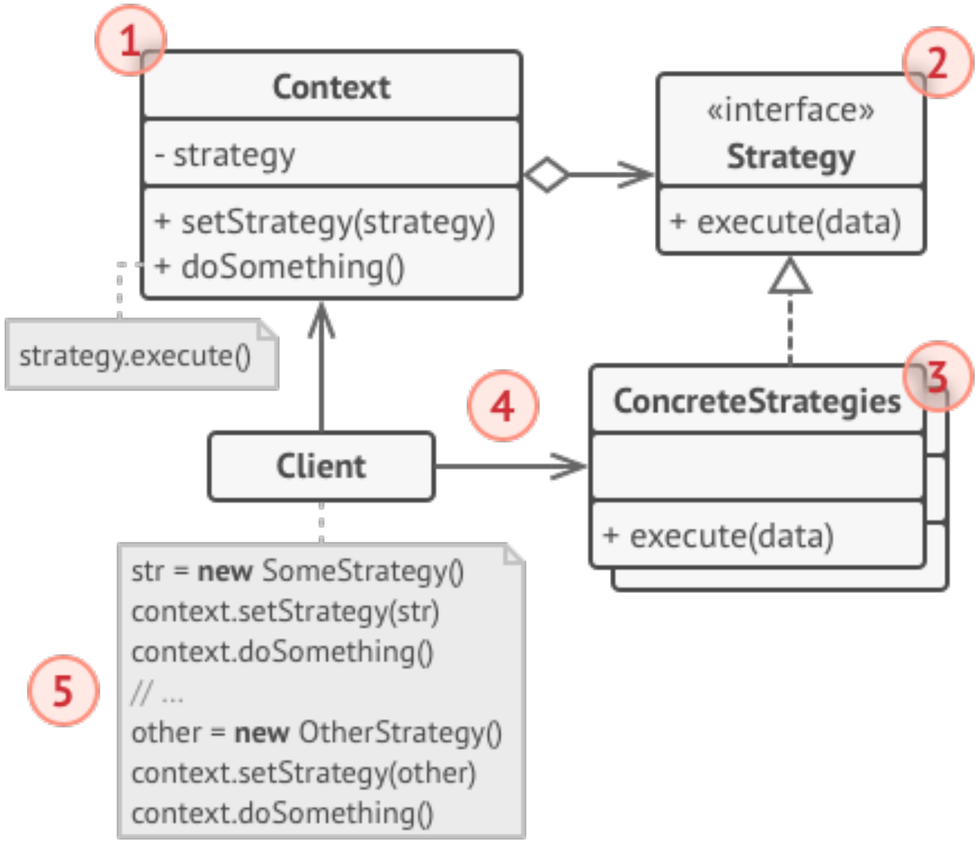
```
public static void main(String[] args) {

    MallardDuck mallard = new MallardDuck();
    RubberDuck  rubberDuckie = new RubberDuck();
    DecoyDuck   decoy = new DecoyDuck();

    Duck     model = new ModelDuck();

    mallard.display();
    mallard.fly();
    mallard.quack();

    rubberDuckie.display();
    rubberDuckie.fly();
    rubberDuckie.quack();

    decoy.display();
    decoy.fly();
    decoy.quack();

    model.display();
    model.fly();
    model.quack();
}
```

Després d'aplicar el patró:

```
public class MiniDuckSimulator {

    public static void main(String[] args) {

        MallardDuck mallard = new MallardDuck();
        RubberDuck  rubberDuckie = new RubberDuck();
        DecoyDuck   decoy = new DecoyDuck();
        Duck     model = new ModelDuck();

        mallard.performQuack();
        rubberDuckie.performQuack();
        decoy.performQuack();

        model.performFly();
        model.setFlyBehavior(new FlyRocketPowered());
        model.performFly();
    }
}
```

```
public class MallardDuck extends Duck {
    public MallardDuck() {

        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();

    }
    public void display() { System.out.println("I'm a real Mallard duck"); }
}
```

5. Com queden els principis?



S: Single Responsability

O: Open Closed

L: Liskov

I: Interface Segregation

D: Dependency

3.5. Patrons de disseny

Propòsit → **Àmbit ↓**	**CREACIÓ**	**ESTRUCTURA**	**COMPORTAMENT**
CLASSE	• Factory method	• class Adapter	• Interpreter • Template method
OBJECTE	• Abstract Factory • Builder • Prototype • **Singleton** • Object pool	• Object Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsability • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

Patró Singleton

- **Singleton** – Assegura una classe que només té una única instància en tota l'aplicació i proporciona l'accés global a aquesta classe.

Patró Singleton (Projecte)

Projecte del campus: singleton

previ :

- VariableGlobal (main) i SenseSingleton.java: No hi ha singleton
- SingletonClient (main) i Singleton.java: No hi ha singleton

stat:

- atribut estàtic dins de la classe Singleton: eager Singleton

classic:

- atribut estàtic dins de la classe Singleton amb allocatació a la primera crida de getInstance: lazy Singleton

threadsafe:

- codi protegit per a la sincronitació de múltiples threads

subclases:

- codi exemple de com derivar Singletons

Chocolote:

- singleton via Enum