Tema 3: Disseny

Anna Puig

Enginyeria Informàtica

Facultat de Matemàtiques i Informàtica,

Universitat de Barcelona

Curs 2019/2020

UNIVERSITAT DE BARCELONA

Temari

3.4. 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

Classificació (GoF)

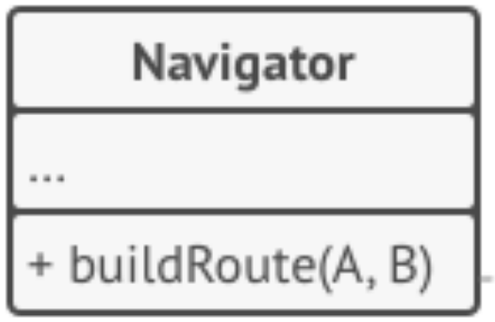
- Creació: Tracten la inicialització i configuració de classes i objectes
  - **Classes**: delega la creació dels objectes a les subclasses
  - **Objectes**: delega la creació d'objectes a altres objectes

- Estructura: Tracten la composició de classes i/o objectes
  - **Classes**: usen herència per composar classes
  - **Objectes**: descriuen maneres d'assemblar objectes

- Comportament: Descriuen situacions de control de flux i caracteritzen el mode en que interactuen i reparteixen responsabilitats les diferents classes o objectes
  - **Classes**: usen herència per descriure els algorismes i fluxos de control
  - **Objectes**: descriuen com un grup d'objectes cooperen per fer una tasca que un objecte la no podria fer sol

Patró Strategy: Problema

Es vol fer un algorisme que calculi la ruta entre un punt A i un punt B del mapa



Patró Strategy

Nom del patró: **Strategy**

Context:

Comportament de diferents lògiques d'un objecte

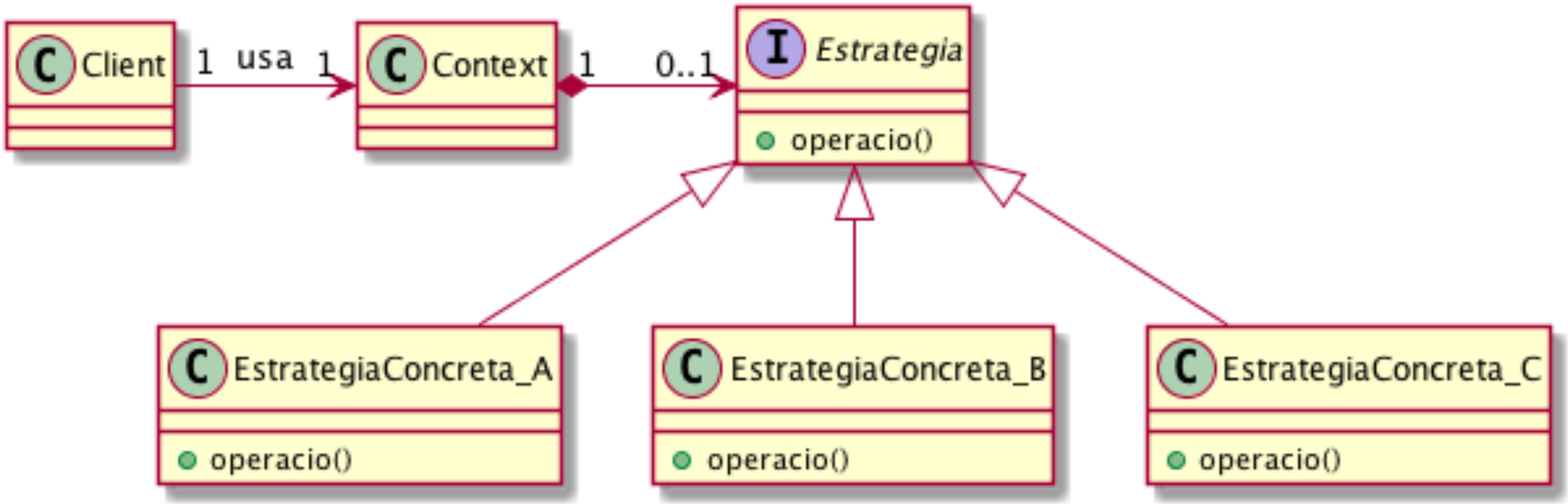
Problema:

- Un mateix problema es pot solucionar amb diferents estratègies o algorismes
- L'elecció d'una de les estratègies pot venir donada pel tipus d'objecte que l'utilitza

Solució:

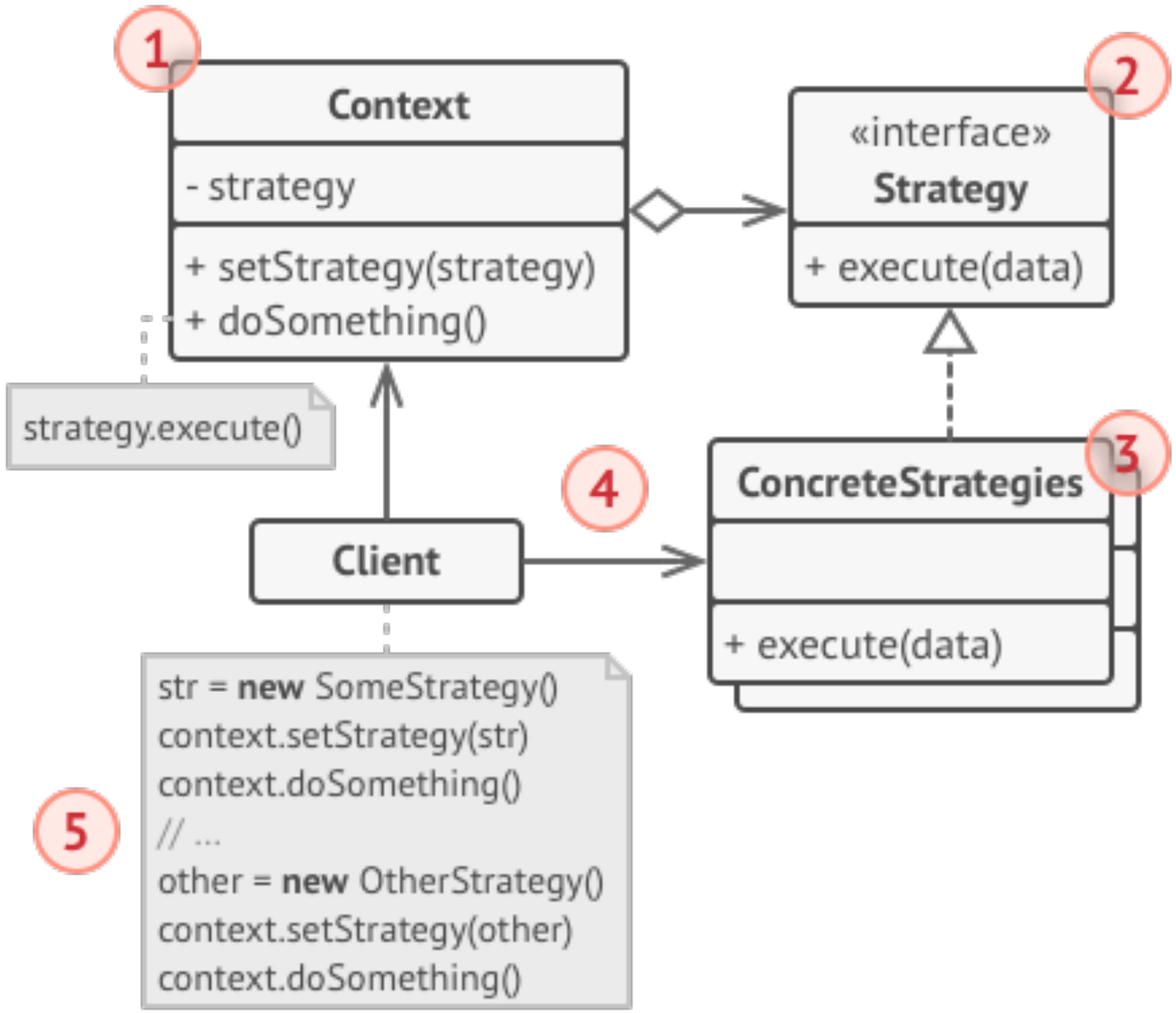
- S'utilitza una interfície comuna per a poder encapsular tots els tipus d'algorismes
- S'utilitzen herències per modelar els clients que utilitzen les diferents estratègies

Patró Strategy



- La classe **`Strategy`** declara la interfície comuna a tots els algorismes
- El **`Client`** es configura amb un context i una estratègia concreta i manté la referència corresponent
- El **`Client`** pot definir una interfície per a que la classe **`Strategy`** accedeixi a les seves dades (i li passarà a l'estratègia la seva referència) o bé li passa totes les dades necessàries en les operacions

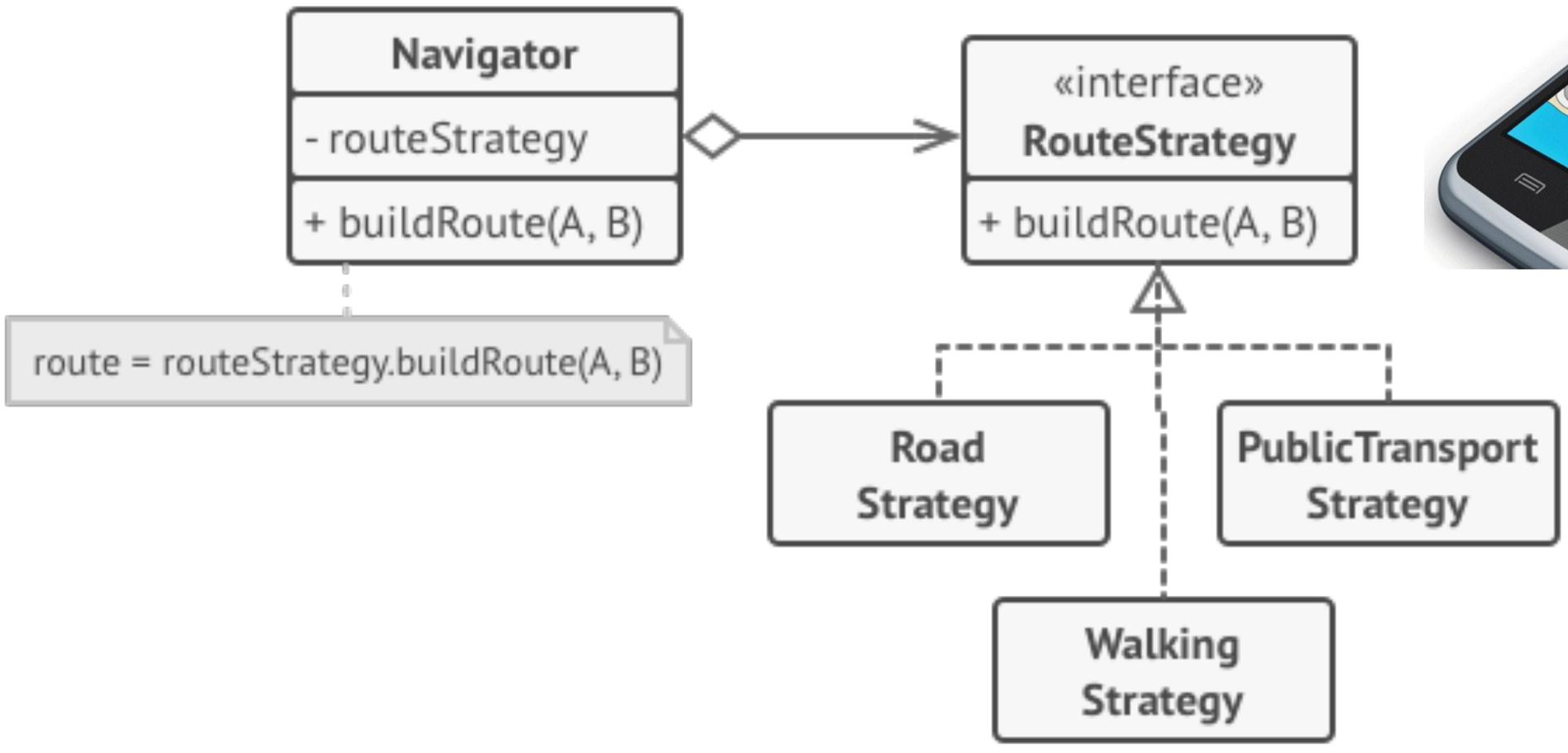
Patró Strategy



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'estratègis concreta que vol fer servir
5. El **`Client`** passa al **`Context`** l'estratègia (setStrategy) i delega en el **`Context`** que l'executi.

Patró Strategy: Solució

Es vol fer un algorisme que calculi la ruta entre un punt A i un punt B del mapa



Patró Strategy

Nom del patró: **Strategy**

Consideracions:

- S'utilitza si es vol modificar l'estratègia a utilitzar en temps d'execució
- S'utilitza quan es tenen classes molt similars que només varien en la forma de comportar-se

Pros:

- S'aïllen els detalls de la implementació de la solució
- Es poden usar diferents estratègies i canviar-les en temps d'execució
- Open-Closed Principle ✓

Cons:

- Els Clients han de tenir clar en què es diferencien les diferents estratègies
- En el cas de tenir pocs algorismes que rarament canvien, no cal complicar més el codi usant aquest patró

Patró Singleton (Idiom)

Nom: **Singleton**

Context

- Assegurar que una classe **només té una sola** instància i proporcionar un punt d'accés global a ella

Problema

- És necessari quan hi ha classes que han de gestionar de manera centralitzada un recurs
  - Un *spooler* d'impressió en un sistema
  - Un gestor de finestres, etc.
  - Controlador
- Una variable **global** no garanteix que només s'instanciï una vegada

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

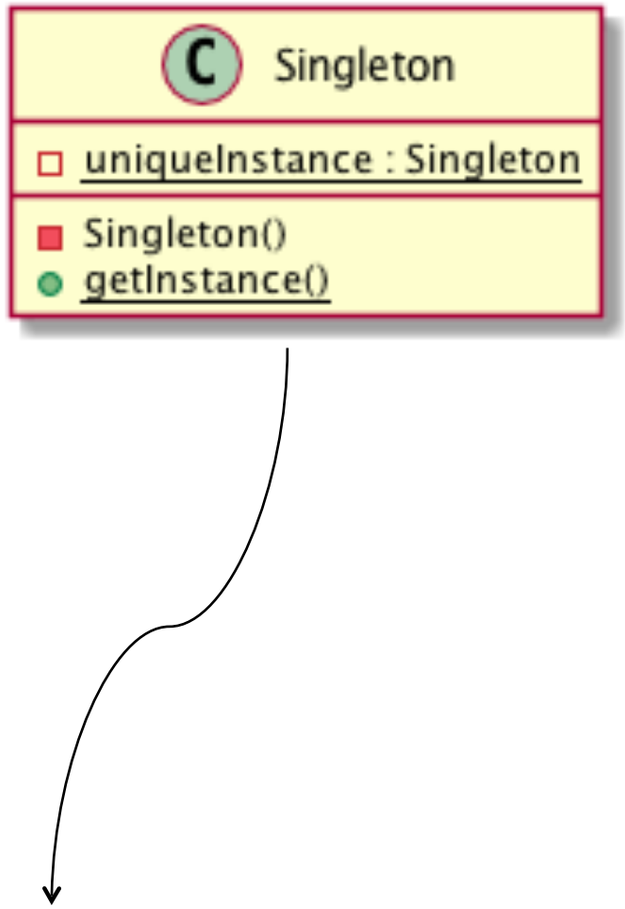
Solució

- És una classe on el constructor és **privat**
- Per a poder-se cridar i que es construeixi:
  - Es fa un mètode static (getInstance())
  - S'afegeix una variable estàtica i privada del mateix tipus de la classe on està continguda (instància)

- No es crea l'objecte fins que és necessari (**Lazy Initialization**)

- S'afegeix el codi necessari per no crear dues instàncies.

SINGLETON's Class Diagram




```
if (instancia== null )
    instancia= new Singleton()
return instancia;
```

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

Nom del patró: **Singleton**

Consideracions:

- S'utilitza si es vol tenir una única instància d'una classe (per exemple d'un controlador, o una fçana o una base de dades)
- S'utilitza Si es vol tenir accés global a unes dades des de diferents llocs de l'aplicació (per exemple: blackboards)

Pros:

- L'objecte Singleton només s'inicialitza la primera vegada que es crida.

Cons:

- Difícil de mantenir en entorns de multithreading
- Pot enmascarar alts acoblaments entre objectes

Patró Singleton: Múltiples threads no òptim

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

   // other useful instance variables here

   private Singleton() {}

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

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

Què passa amb múltiples threads?

Patró Singleton: Múltiples threads pseudo-òptim

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

   private Singleton() {}

   public static Singleton getInstance() {
      if (uniqueInstance == null) {
         synchronized (Singleton.class) {
            if (uniqueInstance == null) {
               uniqueInstance = new Singleton();
            }
         }
      }
      return uniqueInstance;
   }
}
```

Volatile per optimitzar

Tots ells tenen problemes amb serialització i amb reflexió !

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: millor opció en Java

```
public enum Singleton {
   INSTANCE(“String concret");

   // atributs privats

   private Singleton(String  text) {
    // news dels atributs que calguin comprovant que són null abans de
    // cridar al seus news
   }

  // other useful methods here
}
```

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

           Singleton singleton Singleton.INSTANCE;

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

Eager initialization, thread-safe, serialitzable.
Només es pot cridar al constructor amb String ja predefinit en el INSTANCE