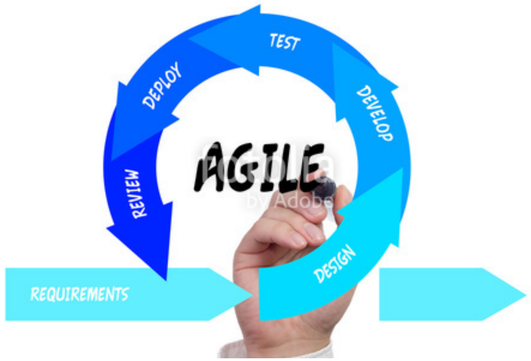


Classe 15.11.2021: Disseny: Intro Patrons de Disseny

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. Introducció a Patrons de Disseny
2. Façana
3. Singleton
4. Strategy
5. Com aplicar **patrons als problemes**?
    - exemple projecte dels Vols

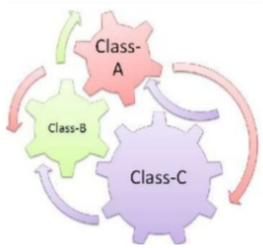
3.3. Criteris GRASP

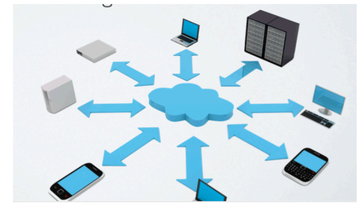
- **Principis generals (GRASP)**: descripció dels principis bàsics d'assignació de responsabilitats a les classes expressades com a patrons (*són molt generals*)

***G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns (GRASP)*

Distribuir responsabilitats és la part més difícil del disseny OO. Consumeix una bona part del temps

- Patrons/Criteris **GRASP**:
  1. Baix Acoblament/Alta Cohesió
  2. Expert en Informació
  3. Creador
  4. Controlador
  5. Fabricació Pura
  6. Indirecció
  7. Polimorfisme
  8. Variacions Protegides





C. Larman
Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (second edition) Prentice-Hall, 2002. (pag. 205 i següents)

3.4. Principis de Disseny: S.O.L.I.D.

Principis de disseny [Robert C. Martin 98]:

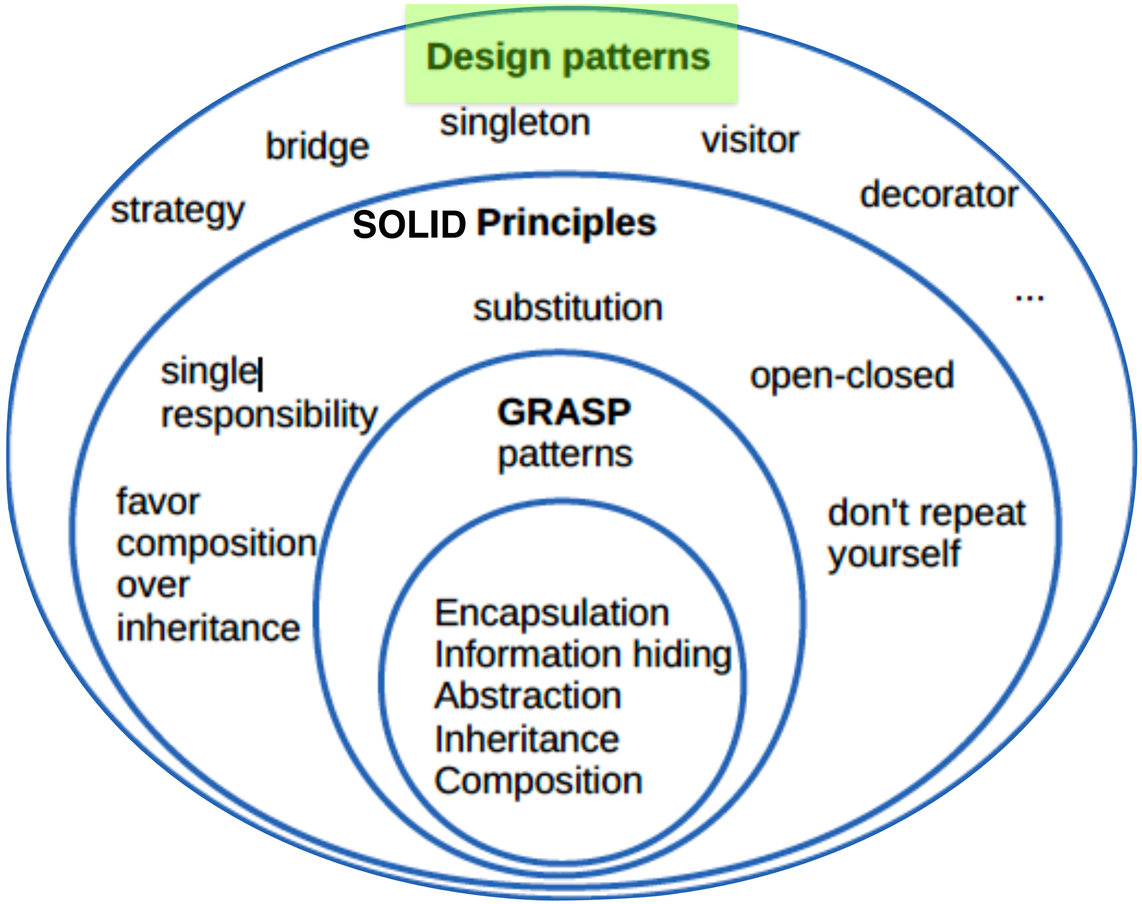
- **S**: Single Responsability Principle
- **O**: Open-Close Principle
- **L**: Liskov Substitution Principle
- **I**: Interface Segregation Principle
- **D**: Dependency Inversion Principle

Article resum de R.C. Martin (fins la pàgina 18)

Agile Software Development, Principles, Patterns, and Practices

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

"*Each **pattern** (patró) describes a problem which occurs over and over again in our environment, and then describes the **core of the solution** to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*"

Christopher Alexander, arquitecte (1977)

Nom del patró

Problema

- Descripció del problema a resoldre
- Enumeració de les forces a equilibrar

Solució

- ***Aspecte estàtic***: impacte en el diagrama de classes del disseny
- ***Aspecte dinàmic***: establiment del comportament de les noves operacions

Conseqüències: Avantatges i desavantatges

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

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

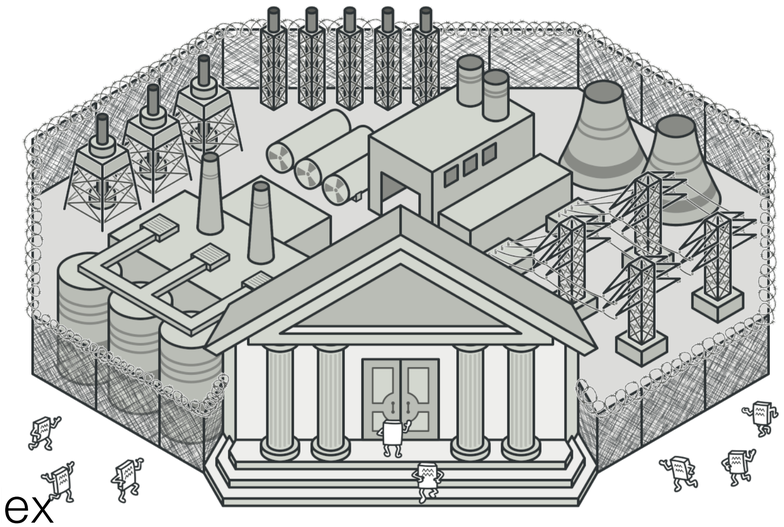
Com utilitzar els patrons?

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

3.5. Patrons de disseny: singleton, facade i strategy

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

3.5.1. Patrons de disseny: Patró Façana (Facade)



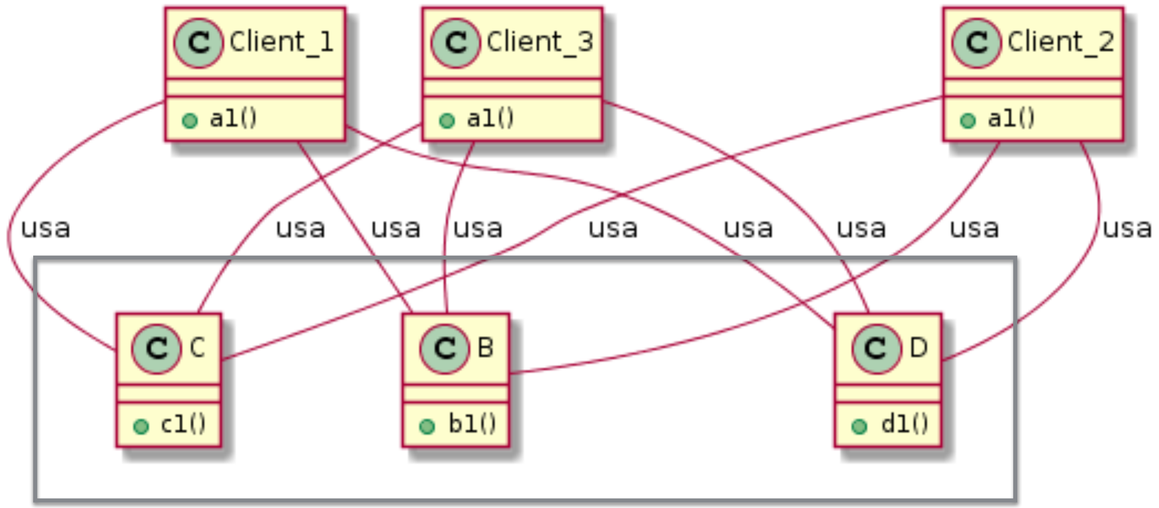
Nom del patró: **Façana** (*Facade*)

Context:

Subsistema amb moltes utilitats diferents

Problema:

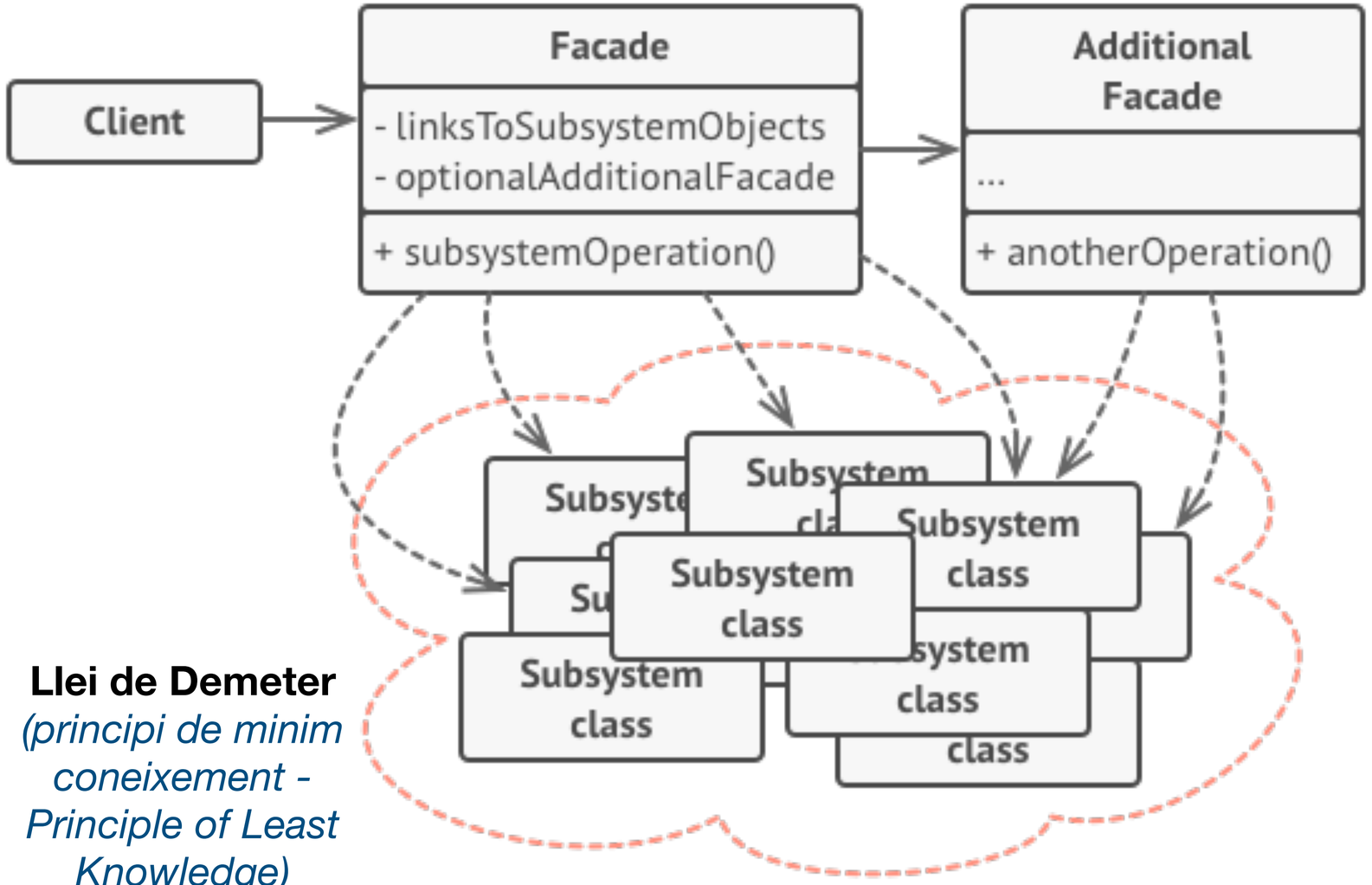
Molt acobalment i complexitat en accedir al subsistema complex



Solució:

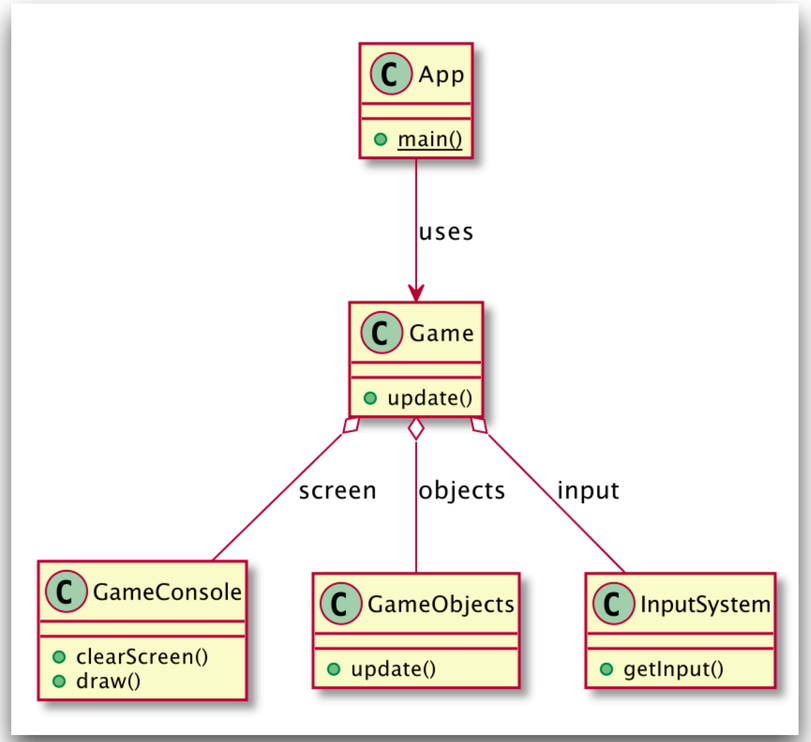
Proporciona una interfície unificada a un conjunt d'utilitats d'un subsistema complex. Redueix la corba d'aprenentatge necessària per aprofitar amb èxit el subsistema.

3.5.1. Patrons de disseny: Patró Façana (Facade)



Llei de Demeter
(principi de minim coneixement - Principle of Least Knowledge)

Projecte del Campus



3.5.2. Patrons de disseny: Patró Façana (Facade)

Conseqüència

- Simplifica l'accés a un conjunt de classes proporcionant una única classe que tots utilitzen per comunicar-se amb el conjunt de classes

Pros:

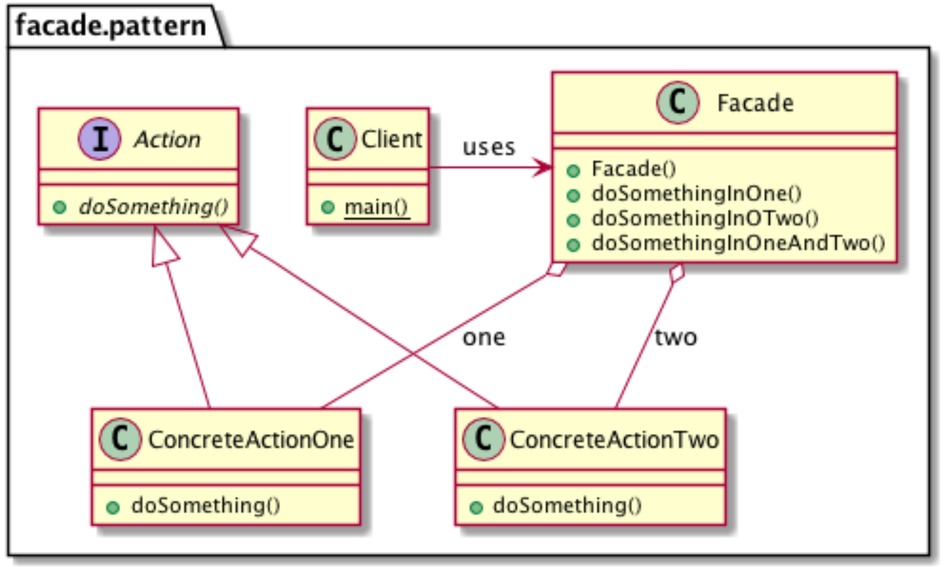
- Les aplicacions client no necessiten conèixer les classes que hi ha darrera la classe FACADE
- Es poden canviar les classes "ocultes" sense necessitat de canviar els clients. Només s'ha de fer els canvis necessaris a FACADE
- Es minimitzen les comunicacions i dependències entre subsistemes

Cons:

- Dóna funcionalitats més limitades que les que realment té el subsistema
- Pot esdevenir un objecte **Deu**

3.5.1. Patrons de disseny: Patró Façana (Facade)

```
public class Facade {

    private ConcreteActionOne one;
    private ConcreteActionTwo two;

    public Facade() {
        System.out.println("This is the FACADE pattern...");
        this.one = new ConcreteActionOne();
        this.two = new ConcreteActionTwo();
    }

    public void doSomethingInOne() {
        System.out.println("Calling doSomething in action ONE:");
        one.doSomething();
    }

    public void doSomethingInOTwo() {
        System.out.println("Calling doSomething in action TWO:");
        two.doSomething();
    }

    public void doSomethingInOneAndTwo() {
        System.out.println("Calling doSomething in action ONE and TWO:");
        one.doSomething();
        two.doSomething();
    }
```



3.5.1. Patrons de disseny: Patró Façana (Facade)

- Reducció de l'acoblament client-subsistema
  - Es pot reduir l'acoblament fent que la **façana** sigui una classe abstracta amb subclasses concretes per les diferents implementacions del subsistema. Els clients es comuniquen amb el subsistema utilitzant la classe façana abstracta
  - Una altra possibilitat és configurar l'objecte façana amb diferents objectes abstractes del subsistema. Per personalitzar la façana només cal canviar un o varis objectes del subsistema
  - El patró **Abstract Factory** es pot utilitzar junt amb la **Facade** per crear objectes del subsistema de manera independent
- Classes del subsistema privades i públiques
  - En Java es pot usar els paquets per determinar les classes que són visibles fora o que no seran visibles.

Exemple: https://springframework.guru/gang-of-four-design-patterns/facade-pattern/

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. 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 (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 (Idiom)

Podem usar una classe estàtica?

- La classe estàtica només pot ser una **classe interna** o una classe aniuada.
- Les classes estàtiques poden utilitzar qualsevol tipus de modificador d'accés (privat, protegit, públic o predeterminat) com qualsevol altre membre estàtic.
- Les classes estàtiques **només poden accedir als membres estàtics** de la classe que la conté.
- La classe estàtica només pot interactuar amb un atribut o mètode no estàtic a través de l'objecte de classe que la conté, ja que **no pot accedir directament als membres no estàtics** de la seva classe que l'envolta.

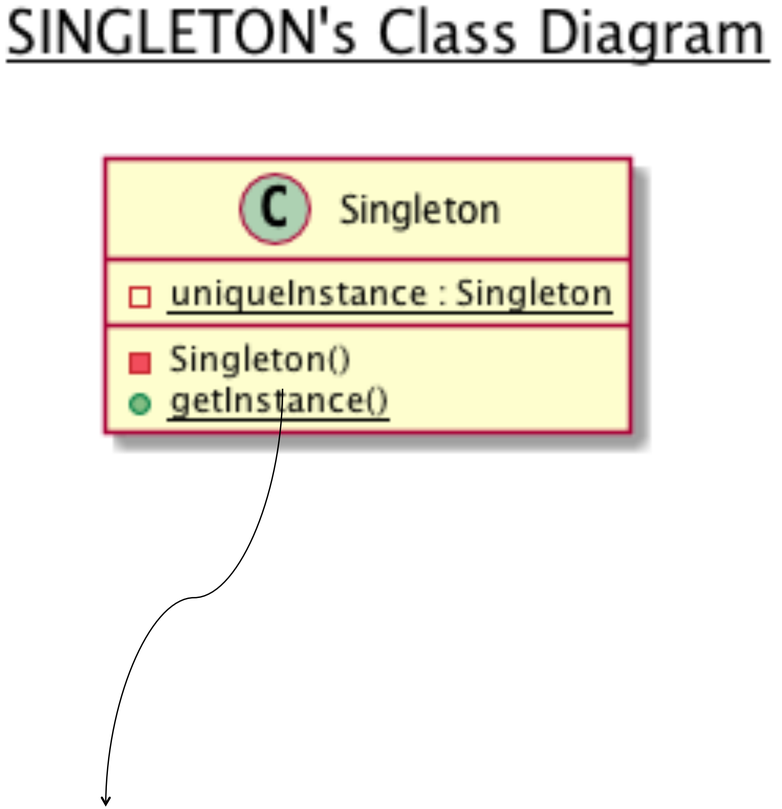
Patró Singleton

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.




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

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 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 façana 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
- **Anti-pattern**
- En alguns casos és millor usar Factories
- Difícil de testejar

Patró Singleton

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