

# ENGINYERIA DEL SOFTWARE II

## Introducció als patrons de disseny (1/2)

GEINF - GDDV

Departament Informàtica i Matemàtica Aplicada  
Escola Politècnica Superior  
Universitat de Girona

Curs 2025/26

IMPORTANT

Aquest document conté les *diapositives* que faig servir a classe. No està pensat ni com uns apunts ni, molt menys, com un capítol d'un *llibre de text*.

Només us ho deixo al Moodle per si us és útil per seguir les classes i caldria que ho complementéssiu amb les explicacions de classe o la bibliografia recomanada.

# Contingut assignatura

- ➊ Enginyeria del software orientada a objectes.
- ➋ Introducció als patrons
  - GRASP (General Responsibility Assignment Software Patterns)
- ➌ Patrons de disseny ([els veurem en un ordre diferent](#))
  - ➍ *Comportament:* Chain of responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.
  - ➎ *Creadors:* Abstract factory, Builder, Factory method, Prototype, Singleton.
  - ➏ *Estructurals:* Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
- ➐ Patrons de patrons (MVC...)

# Taula de continguts

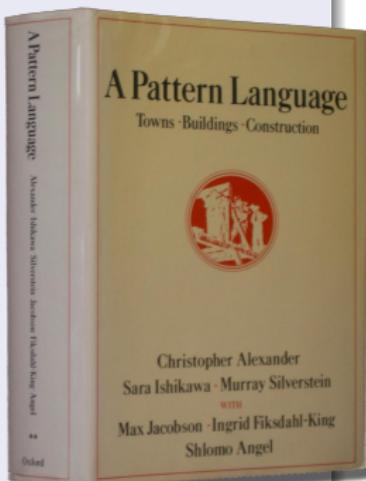
1 Els patrons de disseny de software

2 Cap a un primer patró...

3 Exercici

# Els orígens

## El llibre del 1.977



The elements of this language are entities called patterns. Each pattern 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.

# Els orígens. Algorítmica bàsica

## Esquema de recorregut

```
1  obtenerPrimer
2  mentre no final fer
3      tractarElement
4      obtenerSeguent
5  fimentre
```

## Esquema de cerca

```
1  obtenerPrimer
2  mentre no final i no trobat fer
3      tractarElement
4      si no trobat llavors
5          obtenerSeguent
6          fisi
7  fimentre
```

# Els orígens. Algorítmica bàsica

## Esquema de recorregut

```
1  obtenerPrimer
2  mentre no final fer
3      tractarElement
4      obtenerSeguent
5  fimentre
```

## Esquema de cerca

```
1  obtenerPrimer
2  mentre no final i no trobat fer
3      tractarElement
4      si no trobat llavors
5          obtenerSeguent
6      fisi
7  fimentre
```

**COMPTE:** encara no estem pas dissenyant aquí!

Exposem, això sí, la solució a un problema sense entrar en la sintaxi d'un llenguatge concret.

Els orígens. Algorítmica bàsica

Un recorregut en C++

```
1 for (const auto &element : contenedor){  
2     processa(element);  
3 }
```

# Els orígens. Algorítmica bàsica

## Un recorregut en C++

```
1 for (const auto &element : contenidor){  
2     processa(element);  
3 }
```

## un altre...

```
1 for (int i=0; i<contenidor.size();i++){  
2     processa(contenidor[i]);  
3 }
```

# Els orígens. Algorítmica bàsica

## Un recorregut en C++

```
1 for (const auto &element : contenidor){  
2     processa(element);  
3 }
```

## un altre...

```
1 for (int i=0; i<contenidor.size();i++){  
2     processa(contenidor[i]);  
3 }
```

## i un altre...

```
1 vector<ClasseElement>::const_iterator it=contenidor.begin();  
2 while(it!=contenidor.end()){  
3     processa(*it);  
4     it++;  
5 }
```

# Els orígens. Algorítmica bàsica

## Una cerca en C++

```
1 for (int i=0; i<contenidor.size() && !trobat ;i++){
2     trobat=processa(contenidor[i]);
3     if(trobat) i--;
4 }
```

# Els orígens. Algorítmica bàsica

## Una cerca en C++

```
1 for (int i=0; i<contenidor.size() && !trobat ;i++){
2     trobat=processa(contenidor[i]);
3     if(trobat) i--;
4 }
```

## una altra...

```
1 vector<ClasseElement>::const_iterator it=contenidor.begin();
2 while(it!=contenidor.end() && !trobat){
3     trobat=processa(*it);
4     if (!trobat)
5         it++;
6 }
```

Els orígens. Algorítmica bàsica

## Anirem més enllà

L'exemple anterior fa referència a conceptes algorítmics. En aquesta assignatura, però pujarem un nivell d'abstracció i ens centrarem en el disseny del programari, **i els patrons ens ajudaran a resoldre problemes de disseny**.

# Els patrons de disseny de software

## La base dels patrons

- Els **patrons de disseny** aprofiten l'experiència i coneixements de disseny previs realitzats per problemes similars.
  - Probablement algú abans haurà solucionat problemes com els nostres (o molt similars).

# Els patrons de disseny de software

## La base dels patrons

- Els **patrons de disseny** aprofiten l'experiència i coneixements de disseny previs realitzats per problemes similars.
  - Probablement algú abans haurà solucionat problemes com els nostres (o molt similars).
- Idea de fons: **No reutilitzem codi; reutilitzem experiència.**
  - El codi és sintaxi (memòria curt termini). Els patrons són conceptes, els podem posar a la memòria de llarg termini del cervell...
  - Cerquem pel cervell abans d'omplir l'editor de l'IDE de codi .



Inside Out (Del revés),  
Pixar, 2015

## Una classificació dels patrons de disseny

## Creadors

Abstreu en el procés d'instanciació, treballant a nivell de classes (jugant amb herència per decidir quina classe s'instancia) o a nivell d'objectes (en temps d'execució els objectes decideixen què crear).

# Una classificació dels patrons de disseny

## Creadors

Abstreuen el procés d'instanciació, treballant a nivell de classes (jugant amb herència per decidir quina classe s'instancia) o a nivell d'objectes (en temps d'execució els objectes decideixen què crear).

## Estructurals

Aquests se centren en com aconseguir que classes i objectes s'ajunten per formar estructures més complexes.

## Una classificació dels patrons de disseny

Creadors

Abstreu en el procés d'instanciació, treballant a nivell de classes (jugant amb herència per decidir quina classe s'instancia) o a nivell d'objectes (en temps d'execució els objectes decideixen què crear).

## Estructurals

Aquests se centren en com aconseguir que classes i objectes s'ajunten per formar estructures més complexes.

## Comportament

Treballen amb algoritmes i assignacions de responsabilitats entre objectes, ajudant a controlar fluxes de control

# Una classificació dels patrons de disseny

## Creadors

- Abstract factory.
- Builder.
- Factory method.
- Prototype.
- Singleton.

## Estructurals

- Adapter.
- Bridge.
- Composite.
- Decorator.
- Facade.
- Flyweight.
- Proxy.

## Comportament

- Chain of responsibility.
- Command.
- Iterator
- Mediator.
- Memento.
- Observer.
- State.
- Strategy.
- Template Method.
- Visitor.

# No reinventem la roda

No ens dediquem a reinventar la roda...

## No reinventem la roda

No ens dediquem a reinventar la roda...

tret que ens dediquem a fer rodes!



## Taula de continguts

- 1 Els patrons de disseny de software
  - 2 Cap a un primer patró...
  - 3 Exercici

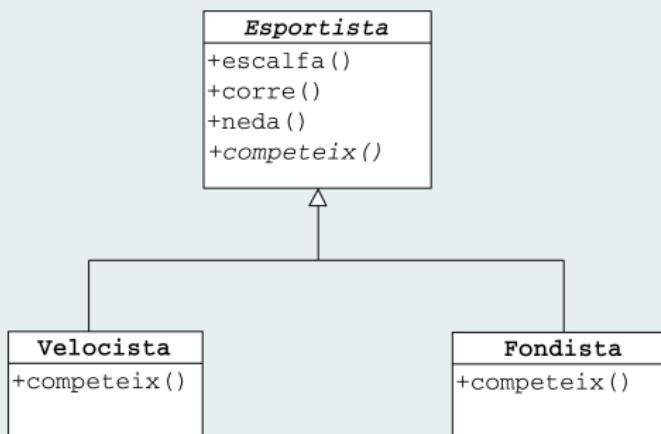
# Disseny esportistes

Volem dissenyar una aplicació on tinguem representats diferents tipus d'esportistes amb mètodes que recullin les seves operacions habituals:

# Disseny esportistes

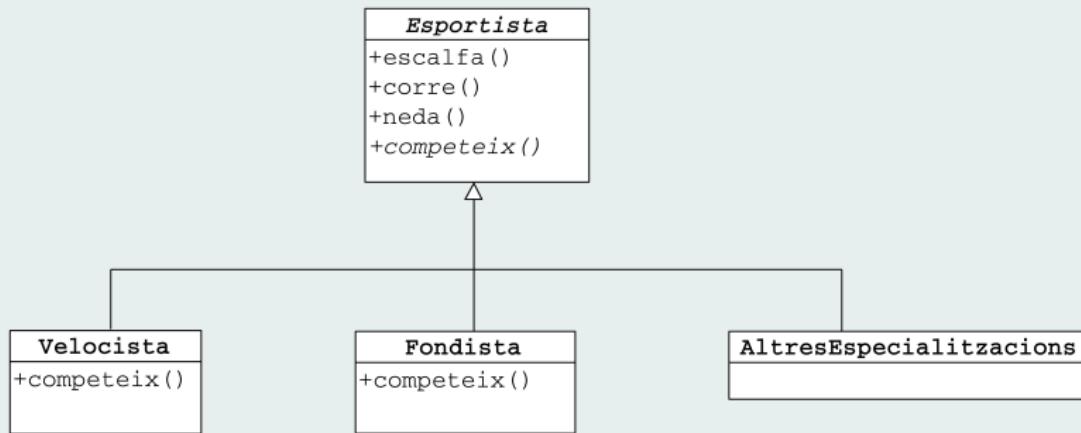
Volem dissenyar una aplicació on tinguem representats diferents tipus d'esportistes amb mètodes que recullen les seves operacions habituals:

## Primer disseny



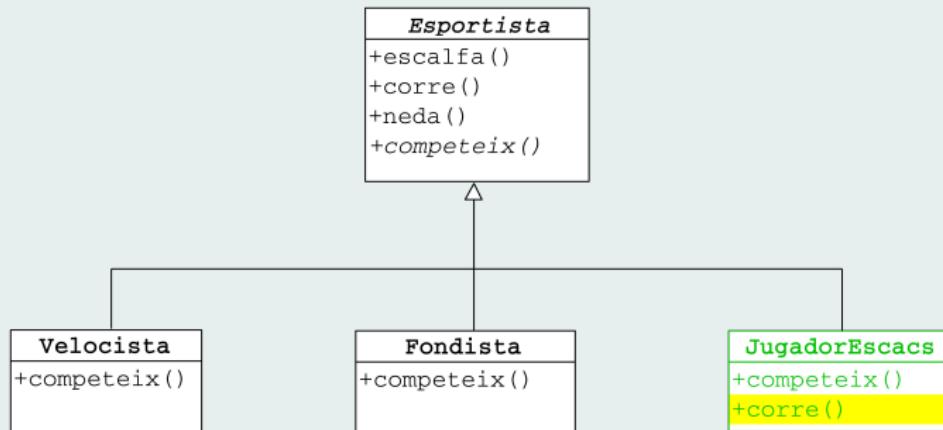
Ampliem el disseny dels esportistes...

Ampliem...



Ampliem el disseny dels esportistes...

Ampliem...



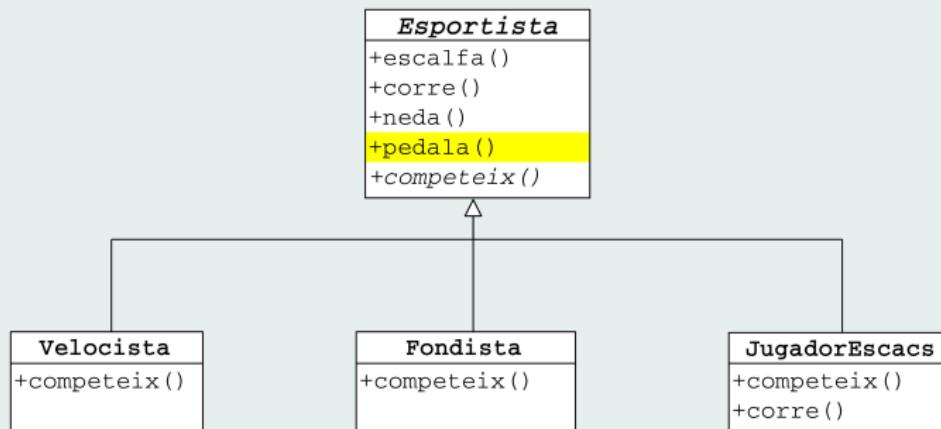
# Els esportistes pedalen...

Els esportistes pedalen? ↪ herència!

Afegim el mètode pedala() a Esportista i problema solucionat.

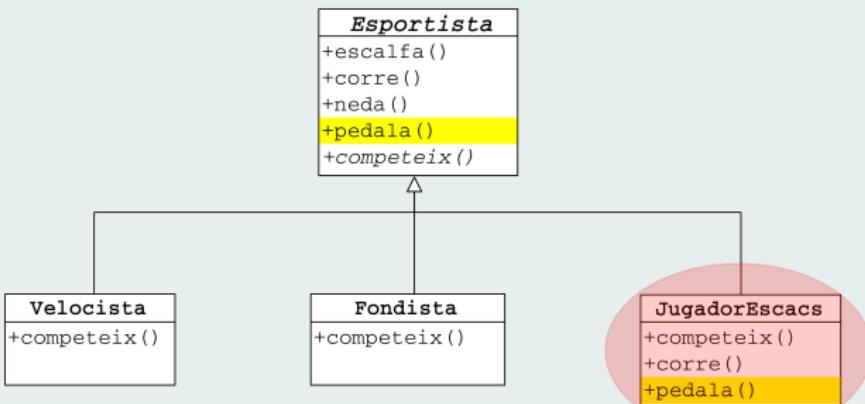
## Els esportistes pedalen...

Ampliem...



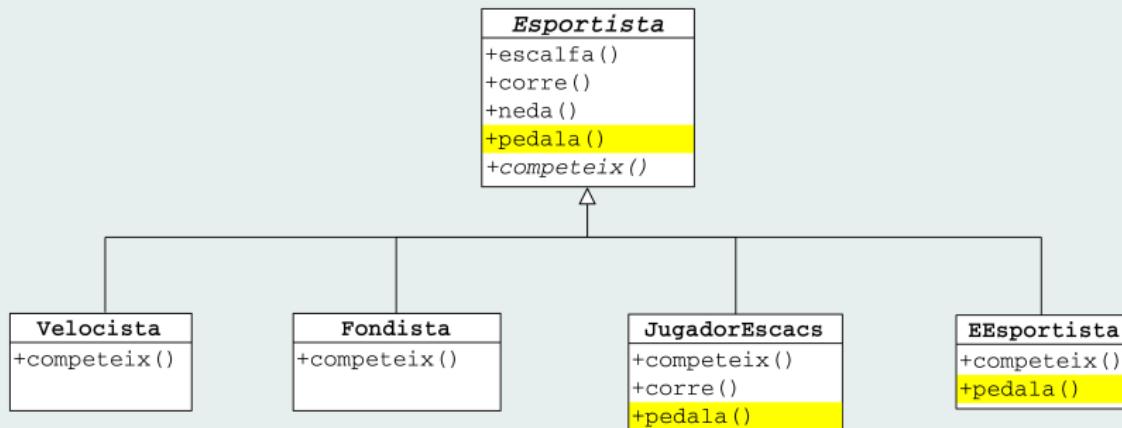
# Els **Alguns** esportistes pedalen...

Ampliem...



# Els **Alguns** esportistes pedalen...

Ampliem...



# Recapitulem...

## Marqueu les que siguin certes

- Tenim codi duplicat entre les classes.
- Canviar els comportaments en temps d'execució és difícil.
- No podem fer fàcilment que els esportistes ballin.
- Els canvis en alguns mètodes poden afectar altres esportistes.
- No podem tenir esportistes que nedin i corrin.
- Tenim dificultats per conèixer el comportament de tots els esportistes.

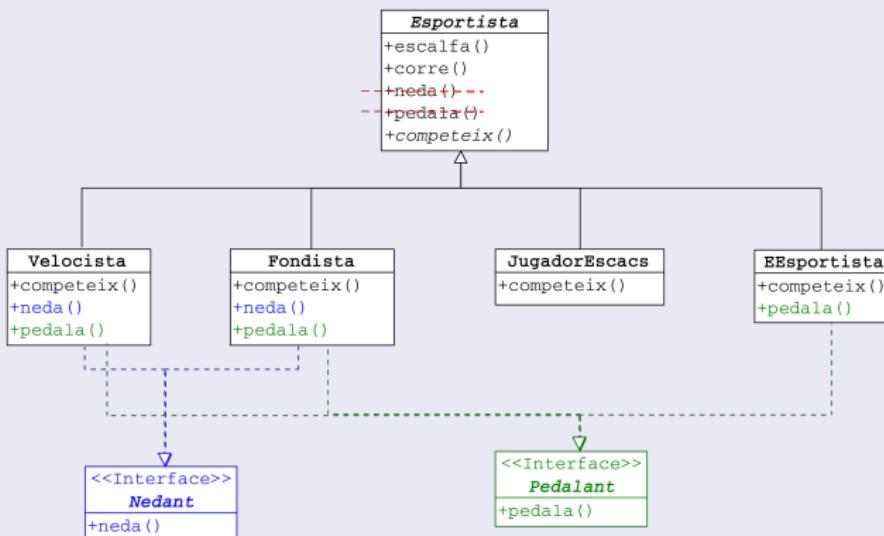
## Recapitulem...

Marqueu les que siguin certes

- Tenim codi duplicat entre les classes.
- Canviar els comportaments en temps d'execució és difícil.
- No podem fer fàcilment que els esportistes ballin.
- Els canvis en alguns mètodes poden afectar altres esportistes.
- No podem tenir esportistes que nedin i corrin.
- Tenim dificultats per conèixer el comportament de tots els esportistes.

# Incorporem interfícies...

Eliminem neda() i pedala() de la superclasse



# Incorporem interfícies...

## Avantatges i inconvenients de fer-ho així

### ↑ Avantatges

**Herència:** Reutilització de codi (només un `neda()` i un `pedala()`).

**Interfícies:** Especificitat (només tenen `neda` i `pedala`) els que ho fan).

### ↓ Inconvenients

**Herència:** Comportament comú a la superclasse... que no és tan comú i obliga a rescriure.

**Interfícies:** No hi ha reutilització de codi; les interfícies només defineixen

Parem un moment...

Quina és la cosa que ens trobarem sempre a l'hora de desenvolupar projectes de software?

# CANVI, EVOLUCIÓ

Parem un moment...

Quina és la cosa que ens trobarem sempre a l'hora de desenvolupar projectes de software?

# CANVI, EVOLUCIÓ

Quins motius de canvi/evolució us venen al cap?  
Escriviu-los aquí: 

# Príncipi de disseny: Encapsular el que varia

- **Agafarem el que pot variar i ho encapsularem** per tal que no afecti a la resta de codi.
  - Obtindrem més flexibilitat a l'hora de modificar el sistema.
  - Reduirem els errors (efectes colaterals) produïts per les modificacions.
- **Encapsular el que varia** és un concepte bàsic recollit per molts dels patrons que anirem veient.

# Príncipi de disseny: Encapsular el que varia

Seguim amb l'exemple dels esportistes

- neda() i pedala() són els mètodes que canvienc...
  - ... els traiem de la jerarquia de classe Esportista

# Príncipi de disseny: Encapsular el que varia

## Seguim amb l'exemple dels esportistes

- `neda()` i `pedala()` són els mètodes que canvienc...
  - ... els traiem de la jerarquia de classe `Esportista`
- Creem dues col·leccions d'algoritmes:
  - *Nedadors* que implementaran diferents maneres de nedar.
  - *Pedaladors* que implementaran diferents maneres de pedalar.

# Príncipi de disseny: Encapsular el que varia

## Seguim amb l'exemple dels esportistes

- `neda()` i `pedala()` són els mètodes que canvienc...
  - ... els traiem de la jerarquia de classe `Esportista`
- Creem dues col·leccions d'algoritmes:
  - *Nedadors* que implementaran diferents maneres de nedar.
  - *Pedaladors* que implementaran diferents maneres de pedalar.
- Dissenyem pensant en les **interfícies**:
  - Una interfície per cada col·lecció (*nedadors*, *pedaladors*).
  - Cada algoritme (`NedaEsquena`, `NedaCroll`, `PedalaPujada`...) implementarà una d'aquestes interfícies.
  - **Les subclasses d'`Esportista` usaran aquests algoritmes.**  
**No els tindran implementats a dins**

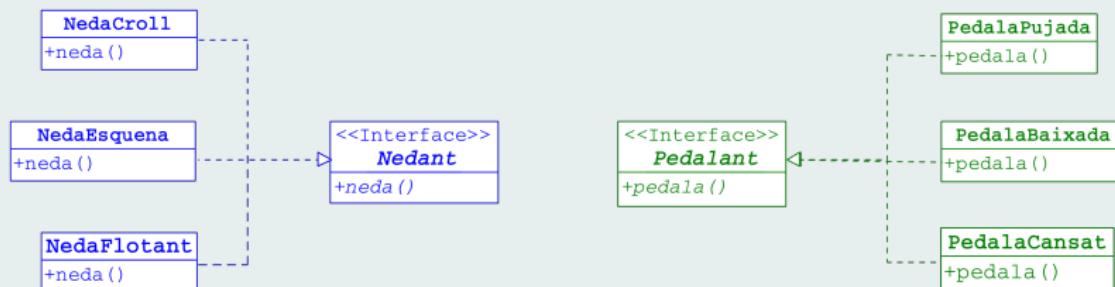
# Les col·leccions d'algoritmes

Ens basem amb el polimorfisme

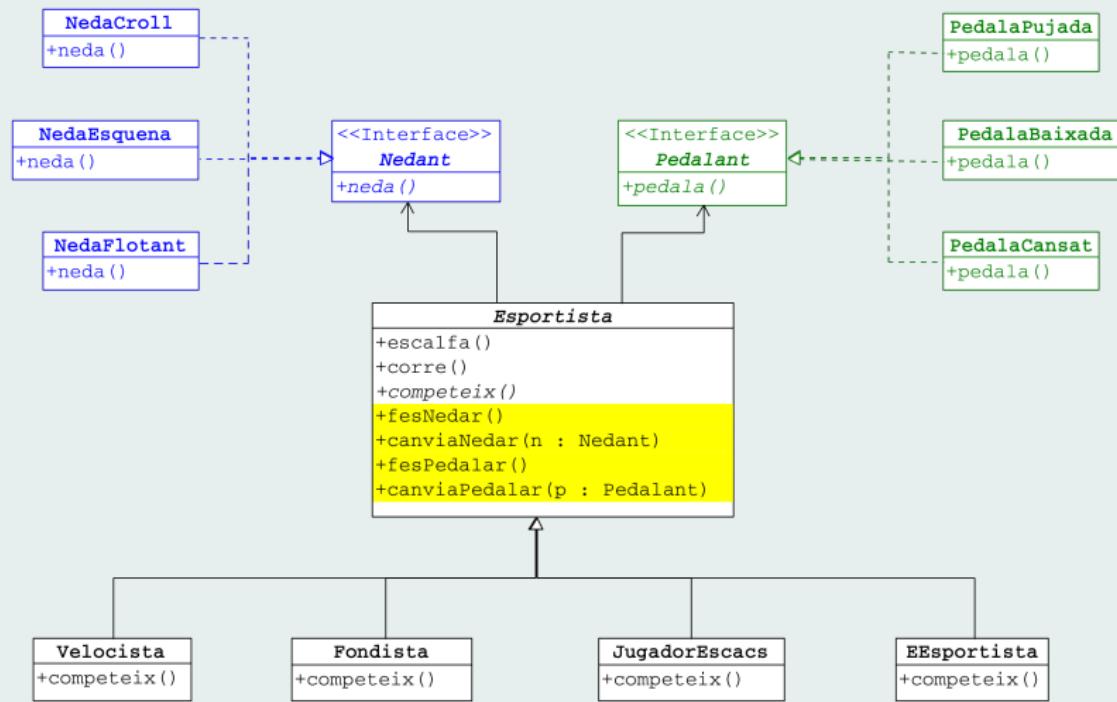
Les implementacions dels algoritmes les podrem tractar com "instàncies" de les interfícies que implementen (*o com les superclasses abstractes pures en cas de C++*)

## Les col·leccions d'algoritmes

Col·leccions d'algoritmes



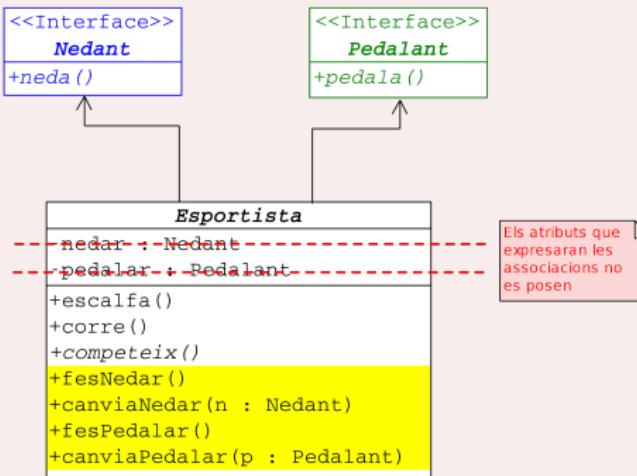
# ... i ajuntant Esportistes i algoritmes...



# ... i ajuntant Esportistes i algoritmes...

## COMPTE!

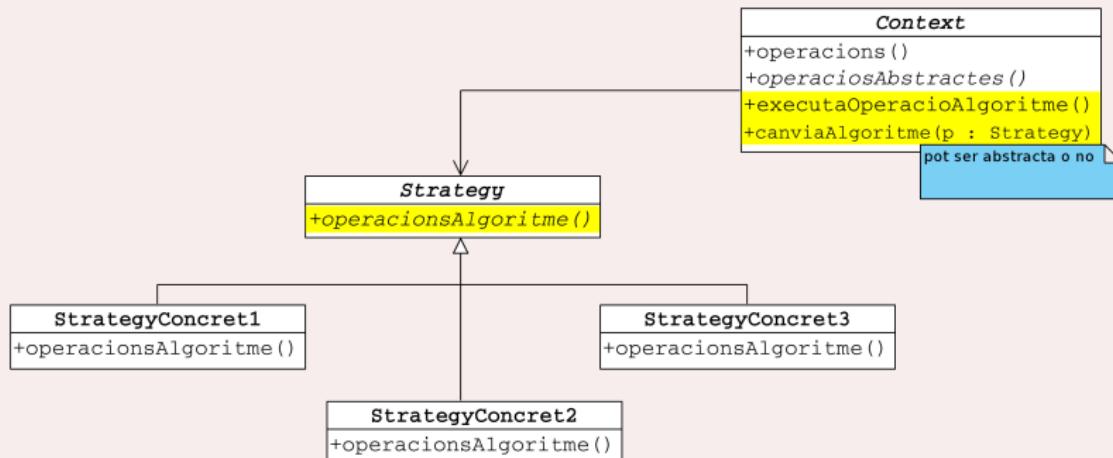
Els atributs de les classes que es faran servir per expressar les associacions no es posen en aquesta fase de disseny.



# Txan! El patró *Strategy*

I ja tenim el primer patró: l'*Strategy*

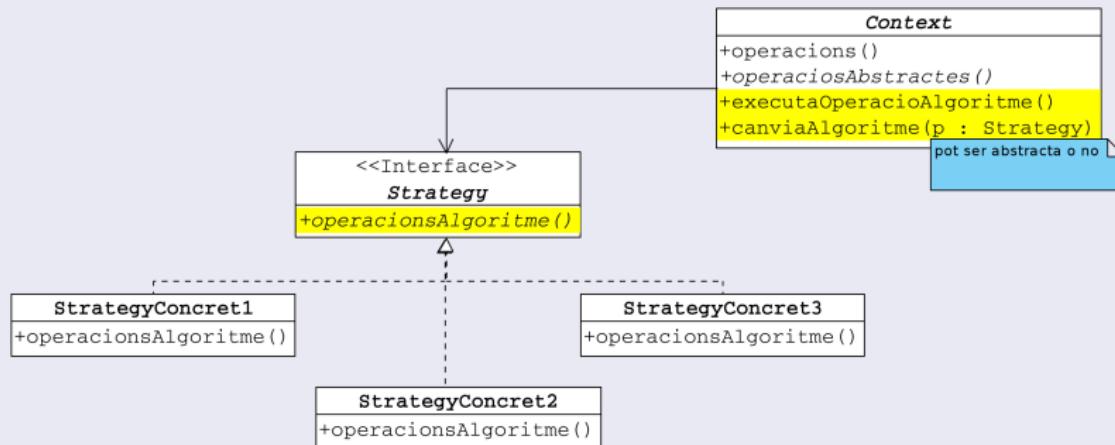
Es pot fer amb herència...



# Txan! El patró *Strategy*

I ja tenim el primer patró: l'*Strategy*

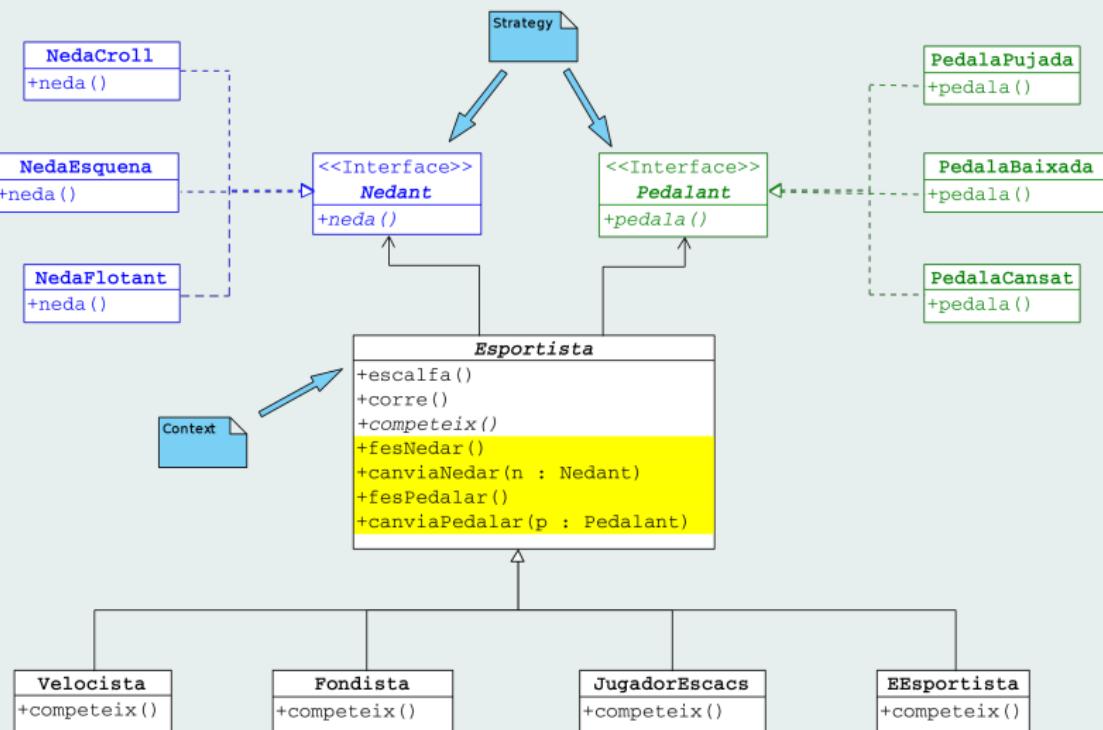
...però **millor amb interfícies.**



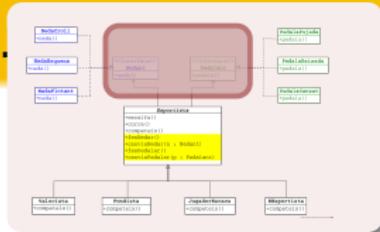
Powered by Apache Protégé Community Edition

# Txan! El patró Strategy

Exemple anterior: qui és Context i qui Strategy



# ... i una senzilla implementació en JAVA...



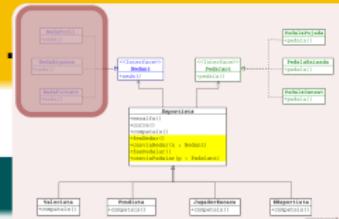
## Nedant

```
1 public interface Nedant {
2     public void neda();
3 }
```

## Pedalant

```
1 public interface Pedalant {
2     public void pedala();
3 }
```

# ... i una senzilla implementació en JAVA...



## Algoritmes nedadors

```
1 public class NedantFlotant implements Nedant{
2     @Override
3     public void neda(){
4         System.out.println("floto_i_prou...");
5     }
6 }
```

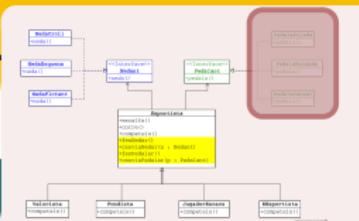
  

```
1 public class NedantCroll implements Nedant{
2     @Override
3     public void neda(){
4         System.out.println("Estic_nedant_croll");
5     }
6 }
```

```
1 public class NedantEsquena implements Nedant{
2     @Override
3     public void neda(){
4         System.out.println("Estic_nedant_esquena");
5     }
6 }
```

# ... i una senzilla implementació en JAVA...



## Algoritmes pedaladors

```
1 public class PedalaPujada implements Pedalant{
2     @Override
3     public void pedala(){
4         System.out.println("pedalant...bufa com puja!");
5     }
6 }
```

```
1 public class PedalaBaixada implements Pedalant{
2     @Override
3     public void pedala(){
4         System.out.println("pedalant...plat gros i cap avall!");
5     }
6 }
```

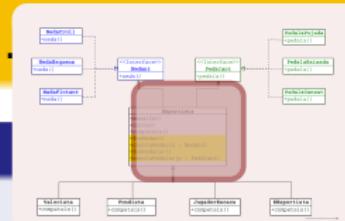
  

```
1 public class PedalaCansat implements Pedalant{
2     @Override
3     public void pedala(){
4         System.out.println("no puc pedalat mes...");
5     }
6 }
```

# ... i una senzilla implementació en JAVA...

## Esportista

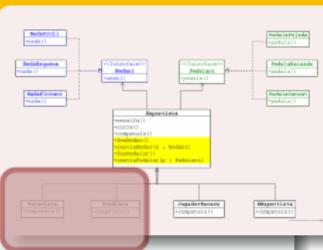
```
1 public abstract class Esportista {  
2     Nedant nedar;  
3     Pedalant pedalar;  
4  
5     public Esportista(){  
6         nedar=new Nedaflotant();  
7         pedalar=new PedalaCansat();  
8     }  
9     public void escalfa(){  
10        System.out.println("Estic_escalfant...");  
11    }  
12    public void corre(){  
13        System.out.println("Estic_corrent...");  
14    }  
15  
16    public abstract void competeix();  
17  
18    public void fesNedar(){  
19        nedar.neda();  
20    }  
21    public void canviaNedar(Nedant n){nedar=n;}  
22  
23    public void fesPedalar(){  
24        pedalar.pedala();  
25    }  
26    public void canviaPedalar(Pedalant p){pedalar=p;}  
27 }
```



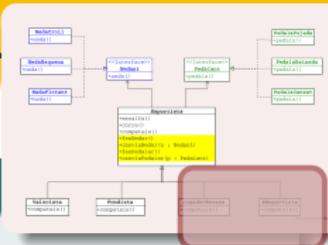
... i una senzilla implementació en JAVA...

## Implementacions d'Esportista

```
1 public class Velocista extends Esportista{  
2  
3     public Velocista(){  
4         nedar = new NedantEsquena();  
5         pedalar = new PedalaCansat();  
6     }  
7  
8     @Override  
9     public void competeix(){  
10        System.out.println("Soc un velocista competint...");  
11    }  
12 }  
  
1 public class Fondista extends Esportista{  
2     public Fondista(){  
3         nedar = new NedantCroll();  
4         pedalar = new PedalaCansat();  
5     }  
6     @Override  
7     public void competeix(){  
8        System.out.println("Soc un fondista competint... no m'apreteu-massa");  
9    }  
10 }
```



... i una senzilla implementació en JAVA...



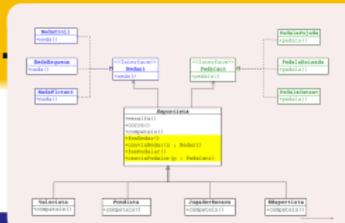
## Implementacions d'Esportista (2)

```
1 public class JugadorEscacs extends Esportista{
2     public JugadorEscacs(){
3         nedar = new NedaFlotant();
4         pedalar = new PedalaCansat();
5     }
6     @Override
7     public void competeix(){
8         System.out.println("Soc un jugador d'escacs pensant... J9");
9     }
10 }
```

```
1 public class EEsportista extends Esportista{
2     public EEsportista(){
3         nedar = new NedaFlotant();
4         pedalar = new PedalaCansat();
5     }
6     @Override
7     public void competeix(){
8         System.out.println("Algu tu un joystick que funcioni be?");
9     }
10 }
```

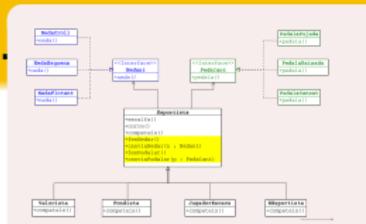
... i una senzilla implementació en JAVA...



## Programa principal

```
1 public class SimuladorStrategy {  
2     public static void main(String[] args){  
3         Esportista vel = new Velocista();  
4         Esportista esc = new JugadorEscacs();  
5         vel.competeix();  
6         vel.fesNedar();  
7         vel.fesPedalar();  
8         esc.competeix();  
9         esc.fesNedar();  
10        esc.canviaNedar(new NedantEsquena());  
11        esc.fesNedar();  
12        esc.canviaNedar(new NedantCroll());  
13        esc.fesNedar();  
14        esc.canviaNedar(new NedaFlotant());  
15        esc.fesNedar();  
16    }  
17 }
```

... i una senzilla implementació en JAVA...



## Resultat main anterior

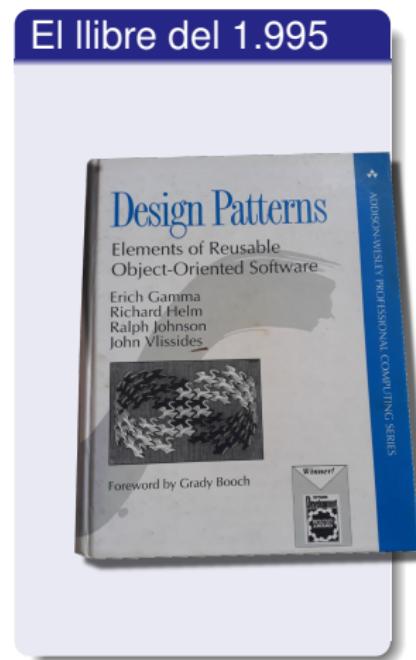
Sóc un velocista competint...  
Estic nedant esquena  
no puc pedalar mes...  
Sóc un jugador d'escacs pensant... J9  
floto i prou...  
Estic nedant esquena  
Estic nedant croll  
floto i prou...

Per acabar, recuperem el llibre dels GoF

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides,  
*Design Patterns: Elements of Reusable Object-Oriented Software*,  
Addison Wesley, 1995

Patrons GOF

Es coneixen com **GOF** (**Gang Of Four**) en referència als 4 autors del llibre



# Per acabar, recuperem el llibre dels GoF

## Els quatre components d'un patró de disseny (segons GoF)

In general, a pattern has four essential elements:

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.
2. The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

# Per acabar, recuperem el llibre dels GoF

## Els quatre components d'un patró de disseny (segons GoF)

3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.
4. The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

# Per acabar, recuperem el llibre dels GoF

Pel patró *Strategy*...

- ① **Nom del patró** *Strategy*.
- ② **Problema** Tenim classes amb comportaments similars i ens interessa que aquests puguin anar canviant en temps d'execució sense haver d'escriure una bateria d'`if then else` i no haver de duplicar codi entre les classes.
- ③ **Solució** Extreure els mètodes de la classe que varien, encapsular-los en una col·lecció d'algoritmes i substituir els mètodes a la classe original.

# Per acabar, recuperem el llibre dels GoF

Pel patró *Strategy*...

## 1 Conseqüències

- ↑ Ens ofereix una alternativa a l'herència per especialitzar classes... tot i que podem fer servir l'herència en el disseny de les classes amb els algoritmes.
- ↑ Aconseguim un codi més clar eliminant tota la bateria de condicionals que necessitaríem per canviar comportaments.
- ↑ Podem oferir alternatives pels mateixos comportaments (més ràpides, amb menys consum de memòria,...).
- ↓ Els clients han de conèixer bé els algoritmes disponibles.
- ↓ No totes les implementacions dels algoritmes poden haver de fer servir els mateixos paràmetres.
- ↓ Hi hauran més classes al disseny i més objectes en temps d'execució.

# Taula de continguts

1 Els patrons de disseny de software

2 Cap a un primer patró...

3 Exercici

# Exercici

## Per fer amb els trios de pràctiques

- Imagineu una situació del món real (o imaginari) en la que es pogués aplicar el patró *Strategy*.
  - Descriviu-la textualment.
  - Feu l'esborrany del diagrama de classe d'aquest disseny.
- Pengeu al Moodle aquest esborrany.
  - En un document en format PDF (assegureu-vos que el diagrama de classes és lleigible)
  - No oblideu posar-hi el nom de tots els membres del grup.
  - **Lliurament: fins el dia que consta al Moodle.**



©Jordi Regincós-Isern [Universitat de Girona], 2025  
jordi.regincos@udg.edu

Aquesta obra està subjecta a una llicència Reconeixement-CompartirIgual 4.0 de Creative Commons