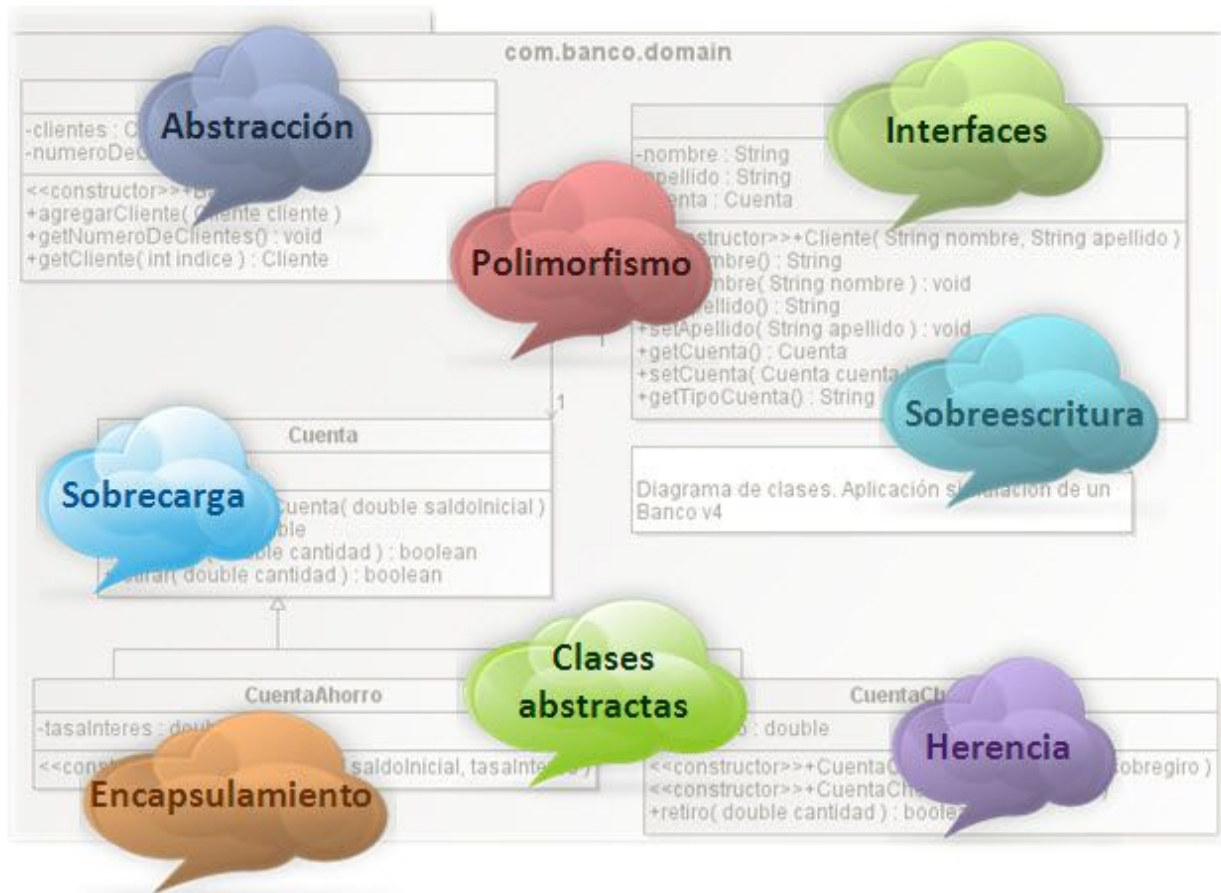


# M03 - UF4 - POO

## Classes i objectes



## 1. PER QUÈ LA POO?

## 2. OBJECTES: propietats i accions

## 3. CLASSES D'OBJECTES: atributs i mètodes

### 3.1. Principis fonamentals del disseny i programació OO

## 4. Definició de CLASSES

### 4.1. Propietats

### 4.2. Mètodes

### 4.3. Constructors

### 4.4. Ús d'objectes d'una classe

### 4.5. Referència a un objecte

### 4.6. Packages

## 5. Ús de patrons

### 5.1. Quins patrons utilitzarem nosaltres? → Patró arquitectònic MVC + patró persistència BO-DAO

## 6. Herència (extends)

### 6.1. Reescritura (override) de mètodes

### 6.2. Modificadors d'accés

### 6.3. Interfaces

### 6.4. Polimorfisme

## 7. Classes genèriques

### 7.1. Mètodes genèrics

## 8. Bibliografia i Webgrafia

# 1. PER QUÈ LA POO?

Les fases del desenvolupament d'una aplicació per agilitzar processos o resoldre problemes generalment són:

1. Estudi previ
2. Anàlisi dels requeriments
3. Disseny
4. Programació i proves (unitàries i d'integració amb la resta de programari)
5. Implantació
6. Manteniment:
  - a. Correcció d'errors
  - b. Implementació de noves funcionalitats

Aquestes fases poden ser més llargues o curtes, segons la metodologia utilitzada (metodologies tradicionals vs metodologies àgils) i el cicle de vida que s'utilitza pel desenvolupament de l'aplicació.

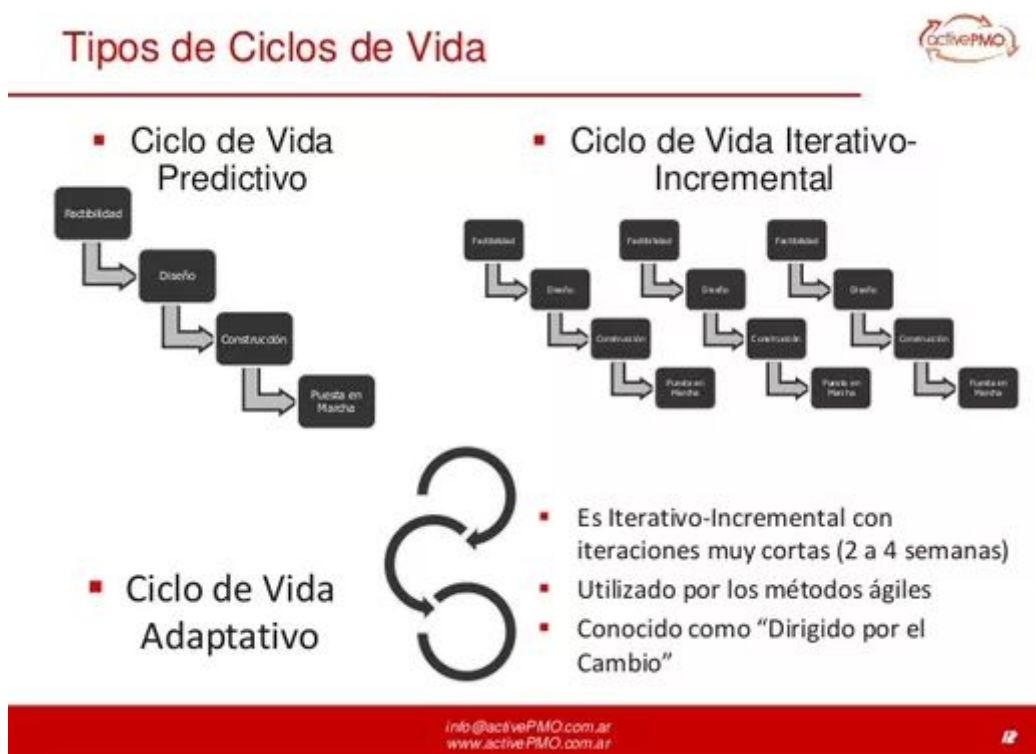
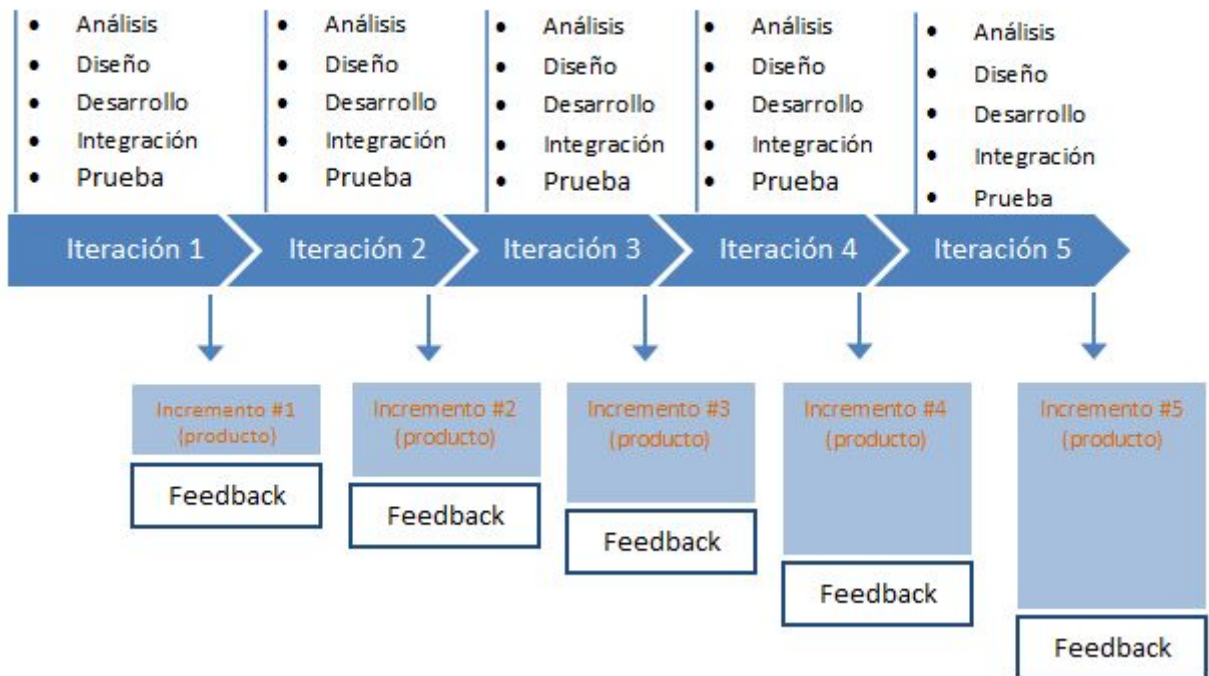


Ilustración 2. Ciclo de vida Ágil de un proyecto

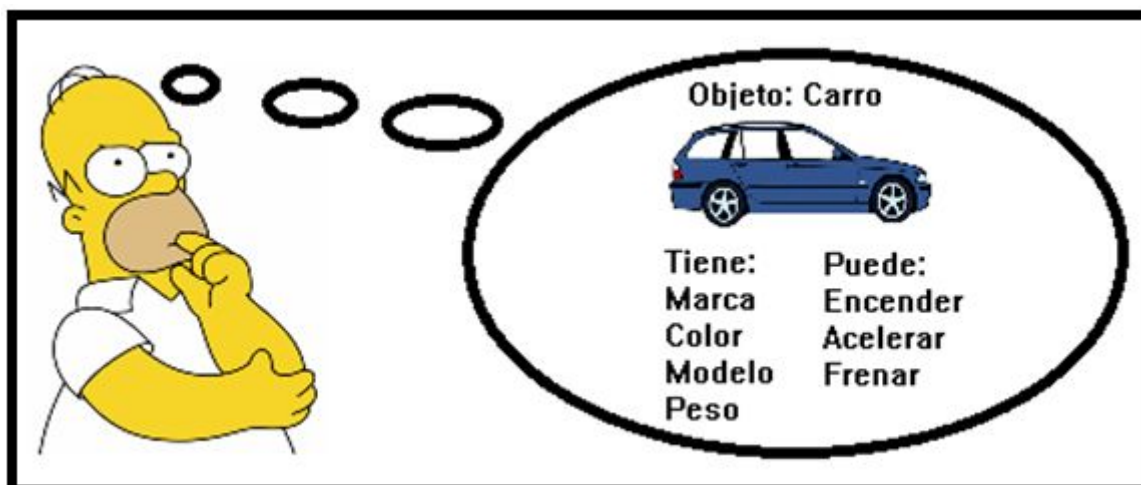


Aplicarem la metodologia OO a les fases de disseny i programació per intentar resoldre els problemes desenvolupant aplicacions d'una manera propera al pensament humà i al món real ⇒ **Aplicacions fàcils de mantenir (escalables) i entendre**

## 2. OBJECTES: propietats i accions

Qualsevol problema es pot plantejar com una simulació d'un escenari del **món real** on intervien diferents objectes, iguals i/o de diferent tipus.

Aquests objectes tenen unes **característiques o propietats** i un realitzen una sèrie **d'accions** que marquen el seu **comportament**.



Quins objectes podem identificar als següents escenaris?





### 3. CLASSES D'OBJECTES: atributs i mètodes

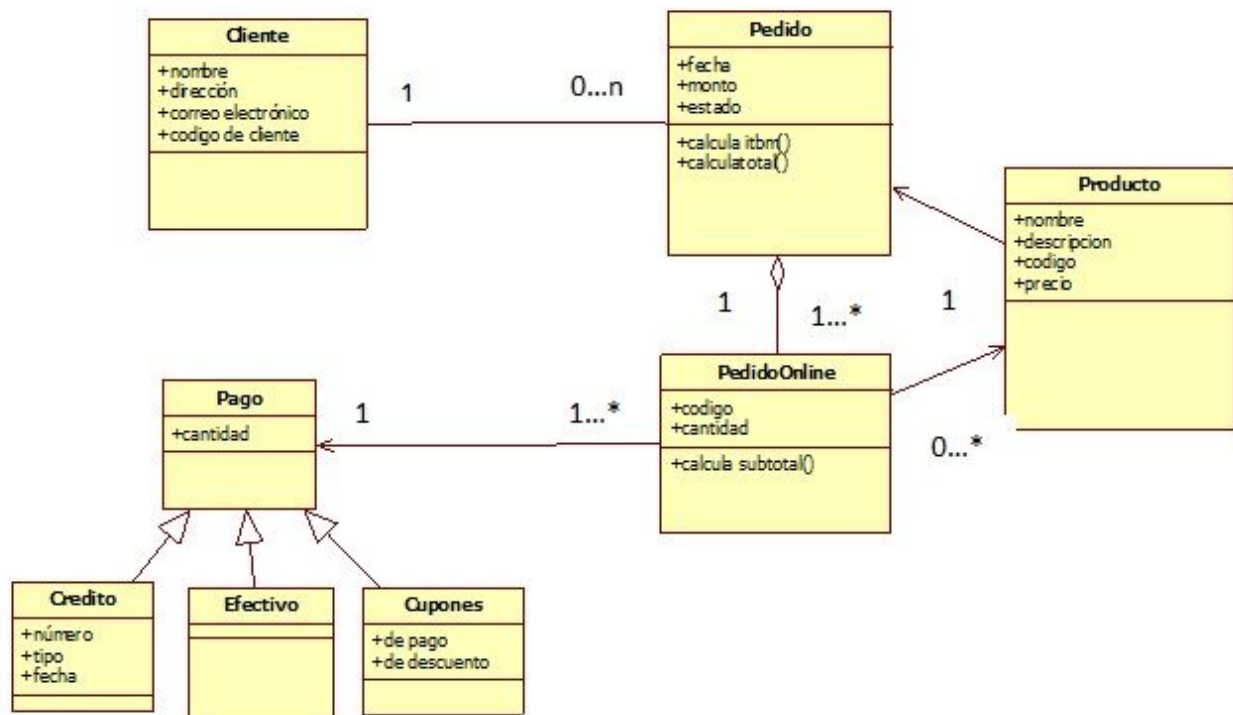
Cada objecte del mateix tipus té les mateixes característiques i comportament, per tant, són del mateix tipus o classe d'objecte. Com els podem representar a un programa? → Utilitzant classes.

Una classe és la definició formal dels **objectes del mateix tipus**. Si parlem en llenguatge informàtic, també podem dir que una classe és la definició d'un nou tipus de dada format per **atributs** (les propietats) i **mètodes** (les accions o comportament).

També podem dir que la definició d'una classe és similar (i algunes vegades fins i tot coincidirà) al d'una taula d'una BD, de la mateixa manera que un objecte d'una classe equival a una fila d'una taula.



Exemple de disseny de classes mitjançant un diagrama de classes:



### 3.1. Principis fonamentals del disseny i programació OO

- **ABSTRACCIÓ**: Capacitat per representar els diferents objectes com a classes.
- **ENCAPSULAMENT**: Cada classe està separada, encapsulada.
  - Implementació interna protegida (ús de modificadors: private, public, etc)
  - Interfície d'ús o API (javadoc)
- **MODULARITAT**: Descomposició del problema en un conjunt de mòduls (packages)
- **JERARQUIA**: Ordenació jeràrquica dels objectes:
  - Ser un tipus de → **GENERALITZACIÓ/ESPECIALITZACIÓ** (Herència) implica **POLIMORFISME**
    - Exemple Herència: Client i Treballador són **un tipus de** Persona. Tenen unes característiques iguals i altres de diferents. Parlem de que Client i Treballador són les classes "filles" i Persona la classe "Mare".
    - Exemple Polimorfisme: Una mateixa persona pot ser un Client o un Treballador, pot agafar diferents "formes". A més podem tractar un Client i un Treballador de la mateixa

manera per determinades coses, per exemple, per calcular la seva edat.

- Ser una part de → **AGREGACIÓ**
  - Exemple: Un Motor **es una part** d'un Cotxe

## 4. Definició de CLASSES


### 4.1. Propietats

```
class Persona {  
    private String nombre;  
    private String apellido1;  
    private String apellido2;  
    private Date fechaNacimiento;  
  
    static int numPersonas = 0;  
}
```

### 4.2. Mètodes

```
class Persona {  
    private String nombre;  
    private String apellido1;  
    private String apellido2;  
    private Date fechaNacimiento;  
  
    static int numPersonas = 0;  
  
    public String getNombre(){  
        return nombre;  
    }  
  
    public void setNombre(String nombre){  
        this.nombre = nombre;  
    }  
  
    public int getEdad() {  
        //càlcul de l'edat segons la data de naixement i la data actual  
        Date fechaActual = new Date();  
        return getEdad(fechaActual);  
    }  
}
```



```
public int getEdad(Date fecha){  sobrecàrrega (overload)  
de mètodes
```

```
    //càlcul de l'edat segons la data de naixement i la data  
    passada com a paràmetre, per saber quina edat tenia o  
    tindrà a una data diferent a l'actual
```

```
    ....
```


```
}
```

```
public static int diferenciaEdad (Persona p1, Persona p2){  
    //càlcul de la diferència d'edat entre dues persones
```


```
    ...
```

```
}
```

```
}
```

 Els arguments o paràmetres d'entrada d'un mètode sempre es passen per VALOR, és a dir, una còpia del valor, però hem de diferenciar entre:

- **Tipus primitius:** si modifiquem la còpia no es modifica el valor de la variable original
- **Tipus objecte:** com que es passa una còpia del identificador (referència) de l'objecte, si modifiquem les propietats de l'objecte, aquestes quedaran modificades també a l'objecte original (perquè apunten al mateix objecte).

 Relacionat amb el punt anterior. Tot i que es poden modificar les propietats d'un objecte passat com a paràmetre, no és habitual fer-ho perquè els fonaments de la POO indiquen que les propietats d'un objecte es modifiquen pel seu comportament, és a dir, amb l'execució dels seus mètodes, i no enviant-lo a un altre mètode per modificar-los.

## **CORRECTE**

```
Persona p = new Persona();
```

```
p.setNombre("Pepe");
```

```
p.setApellido1("Pérez");
```

```
...
```

## **INCORRECTE**

```

Persona p = new Persona();

ponerNombrePersona(p, "pepe");
ponerApellido1Persona(p, "Pérez");
...

```

### 4.3. Constructors

```

class Persona {
    private String nombre;
    private String apellido1;
    private String apellido2;
    private Date fechaNacimiento;

    static int numPersonas = 0;

    public Persona (){
        nombre = "";
        apellido1 = "";
        apellido2 = "";
        ...
        numPersonas++;
    }

    public Persona (String nombre, String apellido1, String
apellido2){
        this.nombre = nombre;
        this.apellido1 = apellido1;
        this.apellido2 = apellido2;
        ...
        numPersonas++;
    }
}

```

### 4.4. Ús d'objectes d'una classe

```

Persona objper1 = new Persona ("Juan", "Palomo", "Pérez");

```

O bé:

```

Persona objper1 = new Persona();
objper1.setNombre("Juan");
objper1.setApellido1("Palomo");
objper1.setApellido2("Pérez");

```

```
Persona objper2 = new Persona();
objper2.setNombre("Pepe");
objper2.setApellido1("Ruiz");
objper2.setApellido2("Morante");
```

## 4.5. Referència a un objecte

Lloc de la memòria on es guarda un objecte, identificat amb un número. Si utilitzem l'operador == estarem comparant aquest número y no les seves propietats, que és el que se suposa que hem de mirar per saber si són iguals o no. Per tant utilitzarem **equals()** o **compareTo()**.

## 4.6. Packages

Sobretot s'utilitzen per agrupar classes relacionades i evitar conflictes entre noms de classes que s'anomenen igual.

Per crear un package i incloure una classe dintre d'aquest:

```
package cat.badia.m03.agenda;
```

```
class Persona {
    ...
}
```



Respecte al nom del package:

- Ha d'anar en minúscula
- És recomanable que sigui únic a Internet, per tant, incloure el domini de l'empresa o institució

Per utilitzar una classe que està en un altre package:

```
import java.io.BufferedReader;
```

```
import java.io.*;  l'asterisc carrega totes les classes public del package
```

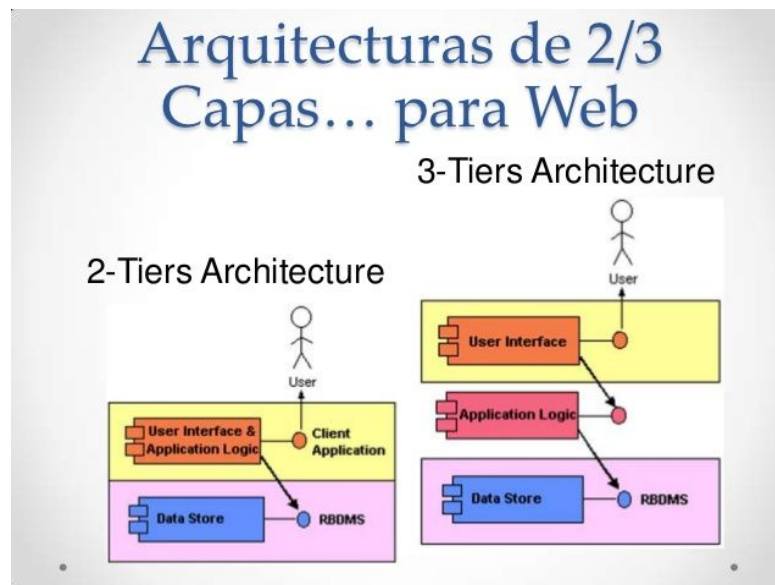
## 5. Ús de patrons

Un patró és una solució **efectiva i reutilitzable** a un problema o desenvolupament **conegut**, i que **es poden combinar entre ells**. Així mateix, existeixen diferents classificacions, com per exemple la següent:

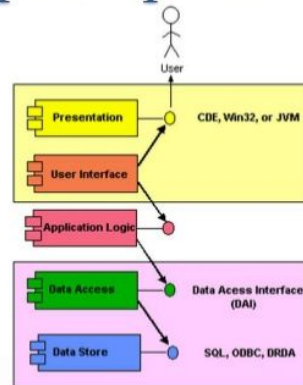
Software design patterns	
<b>Creational</b>	Abstract factory · Builder · Dependency Injection · Factory method · Lazy initialization · Multiton · Object pool · Prototype · RAI · Singleton
<b>Structural</b>	Adapter · Bridge · Composite · Decorator · Delegation · Facade · Flyweight · Front controller · Marker interface · Module · Proxy · Twin
<b>Behavioral</b>	Blackboard · Chain of responsibility · Command · Interpreter · Iterator · Mediator · Memento · Null object · Observer · Servant · Specification · State · Strategy · Template method · Visitor
<b>Functional</b>	Closure · Currying · Function composition · Functor · Monad · Generator
<b>Concurrency</b>	Active object · Actor · Balking · Barrier · Binding properties · Coroutine · Compute kernel · Double-checked locking · Event-based asynchronous · Fiber · Futex · Futures and promises · Guarded suspension · Immutable object · Join · Lock · Messaging · Monitor · Nuclear · Proactor · Reactor · Read write lock · Scheduler · Thread pool · Thread-local storage
<b>Architectural</b>	ADR · Active record · Broker · Client-server · CBD · DAO · DTO · DDD · ECB · ECS · EDA · Front controller · Identity map · Interceptor · Implicit invocation · Inversion of control · Model 2 · MOM · Microservices · MVA · MVC · MVP · MVVM · Monolithic · Multitier · Naked objects · ORB · P2P · Publish-subscribe · PAC · REST · SOA · Service locator · SN · SBA · Specification
<b>Cloud Distributed</b>	Ambassador · Anti-Corruption Layer · Bulkhead · Cache-Aside · Circuit Breaker · CQRS · Compensating Transaction · Competing Consumers · Compute Resource Consolidation · Event Sourcing · External Configuration Store · Federated Identity · Gatekeeper · Index Table · Leader Election · MapReduce · Materialized View · Pipes · Filters · Priority Queue · Publisher-Subscriber · Queue-Based Load Leveling · Retry · Scheduler Agent Supervisor · Sharding · Sidecar · Strangler · Throttling · Valet Key
<b>Other</b>	Business delegate · Composite entity · Intercepting filter · Lazy loading · Mangler · Mock object · Type tunnel · Method chaining
<b>Books</b>	<i>Design Patterns</i> · <i>Enterprise Integration Patterns</i>
<b>People</b>	Christopher Alexander · Erich Gamma · Ralph Johnson · John Vlissides · Grady Booch · Kent Beck · Ward Cunningham · Martin Fowler · Robert Martin · Jim Coplien · Douglas Schmidt · Linda Rising
<b>Communities</b>	The Hillside Group · The Portland Pattern Repository
<b>Authority control</b>	GND: 4546895-3 · LCCN: sh98003823

Anem a veure alguns dels patrons aplicables a la POO o on la POO es veu afectada:

- **Patrons arquitectònics:** defineixen l'arquitectura global d'una aplicació.
  1. Patró Layers i N-tier → patró d'alt nivell on es decideix el número de capes que tindrà la nostra aplicació i la interacció entre elles.

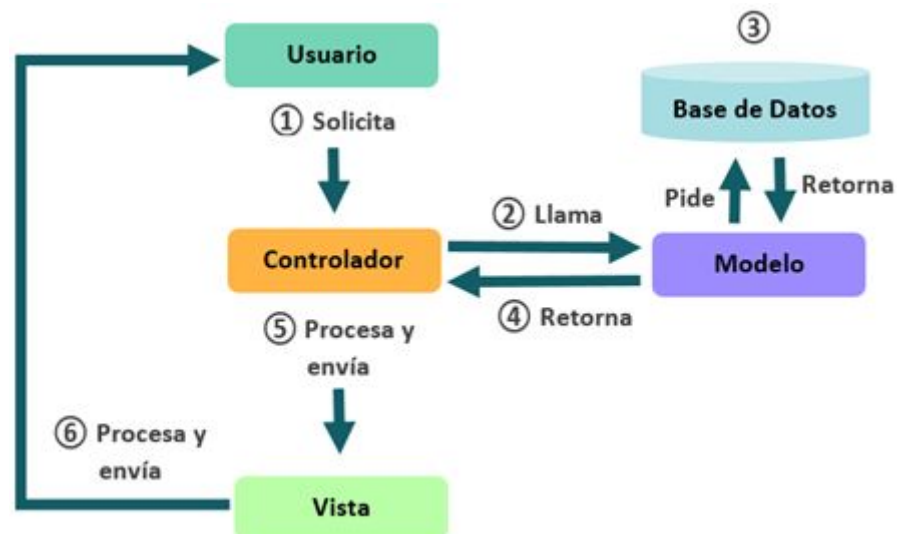


## Arquitectura de 5 capas...para Web



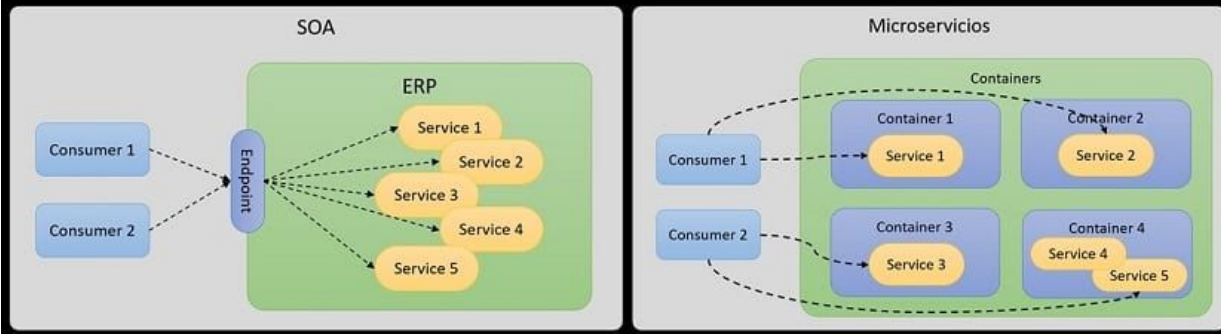
2. Patrón MVC (Model-View-Controller) → patrón derivat de l'anterior, centrat en la interacció amb l'usuari:

- Model: persistència de dades i lògica de negoci (funcionalitats)
- Vista (interfície d'usuari): classes que proporcionen les opcions d'ús de l'aplicació per l'usuari, ja sigui en consola o amb finestres.
- Controlador (interfície d'usuari): classes que reben els **events** (accions) de l'usuari, criden al model per trobar la solució i finalment mostren la solució cridant a la vista



3. Microservices → evolució de l'arquitectura SOA, utilitzada per facilitar el desplegament i escalabilitat de les aplicacions.

# SOA vs Microservicios

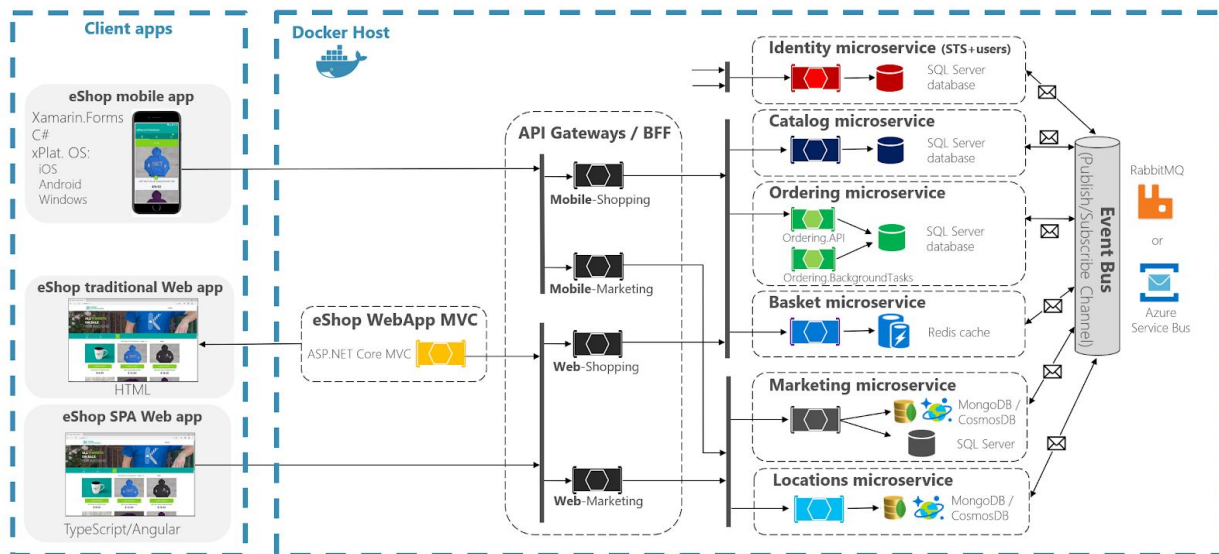


## ESQUEMA DE TRABAJO

Exemple aplicació amb arquitectura microserveis:

### eShopOnContainers reference application

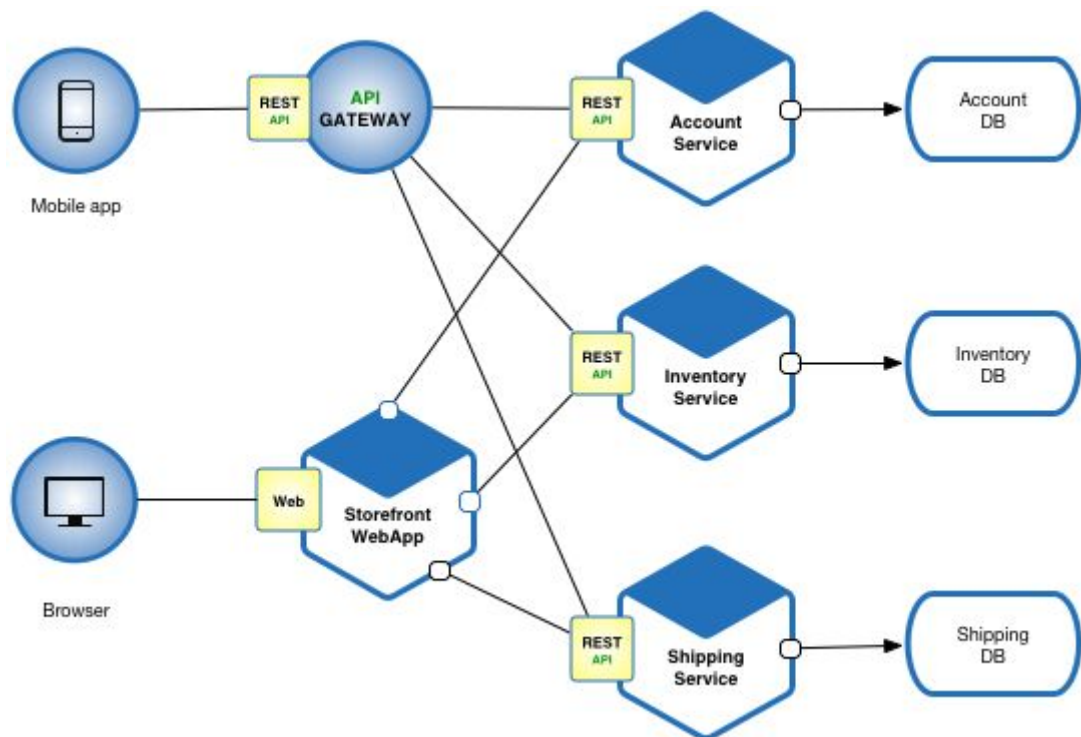
(Development environment architecture)



Un altre exemple:



<https://microservices.io/patterns/microservices.html>



- **Patrons de creacionals, estructurals, etc:** conjunt de patrons utilitzats per resoldre problemes més concrets de desenvolupament de software.

Ex: [Composite](#), que facilita la creació d'estructures d'arbres, com pot ser un explorador d'arxius

Ex: [Abstract Factory](#), utilitzat per realitzar [Dependency Injection](#), un altre patró de disseny.

[https://es.wikipedia.org/wiki/Google\\_Guice](https://es.wikipedia.org/wiki/Google_Guice)

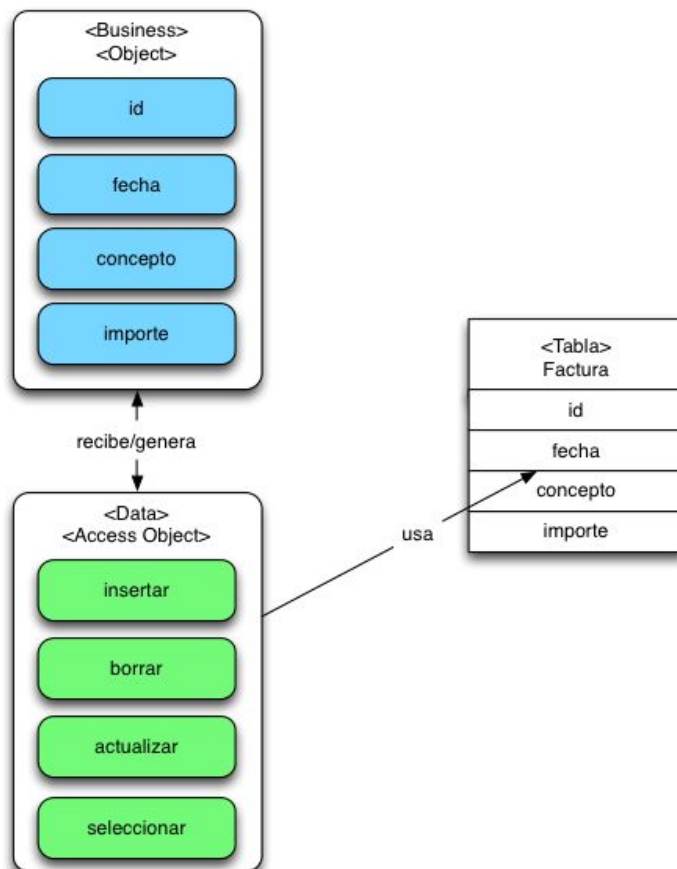
<https://gluonhq.com/labs/ignite/>

- **Patrons de persistència:** defineixen com resoldre el problema concret de com desenvolupar la persistència de la informació de la nostra aplicació. Per exemple, el disseny del Model del punt anterior, i concretament per a la gestió de la persistència es pot fer utilitzant patrons com: Active Record (AR), Bussiness Object/[Value Object](#)/[POJO](#)/[Bean](#) + Data Access Object (BO + DAO) o [altres](#).



<Tabla> Factura
id
fecha
concepto
importe

VS



- **Patrons llenguatge:** defineixen com resoldre un problema concret en una tecnologia o llenguatge de programació concret.

1. Comparació objectes d'una classe en JAVA redefinint el mètode **equals()** de la classe **Object** (classe de la que deriven totes les classes de JAVA).

Exemple: Dos objectes de la classe Persona seran iguals si tots dos tenen el mateix DNI.

```
public class Persona {  
    ...  
    ...  
    public boolean equals(Persona persona){  
  
        boolean iguales = false;  
  
        if ((this.DNI != null) && (persona.getDNI() != null)){  
            if(this.DNI.equalsIgnoreCase(persona.getDNI())){  
                iguales = true;  
            }  
        }  
        return iguales;  
    }  
    ...  
}
```

I el podem utilitzar a la nostra aplicació d'aquesta manera:

```
Persona p1 = new Persona("111111", "Manuel");  
Persona p2 = new Persona("222222", "Jordi");  
  
if(p1.equals(p2)){  
    ...  
}
```

## **5.1. Quins patrons utilitzarem nosaltres? → Patró arquitectònic MVC + patró persistència BO-DAO**

**Model** → Una classe BO + una classe DAO per cada tipus d'objecte que s'hagi d'emmagatzemar (persistència) a l'aplicació. [Exemple](#)

**Vista** → Una classe on es generi la IU per interactuar amb l'usuari.

**Controlador** → Una classe que escolti les diferents accions de l'usuari (opció seleccionada, botó polsat, etc)



Quan la interfície d'usuari sigui la consola, podem posar el controlador i la vista a la mateixa classe, ja que la separació entre vista i controlador no és tant clar que aporti tants beneficis com a una aplicació amb finestres (com

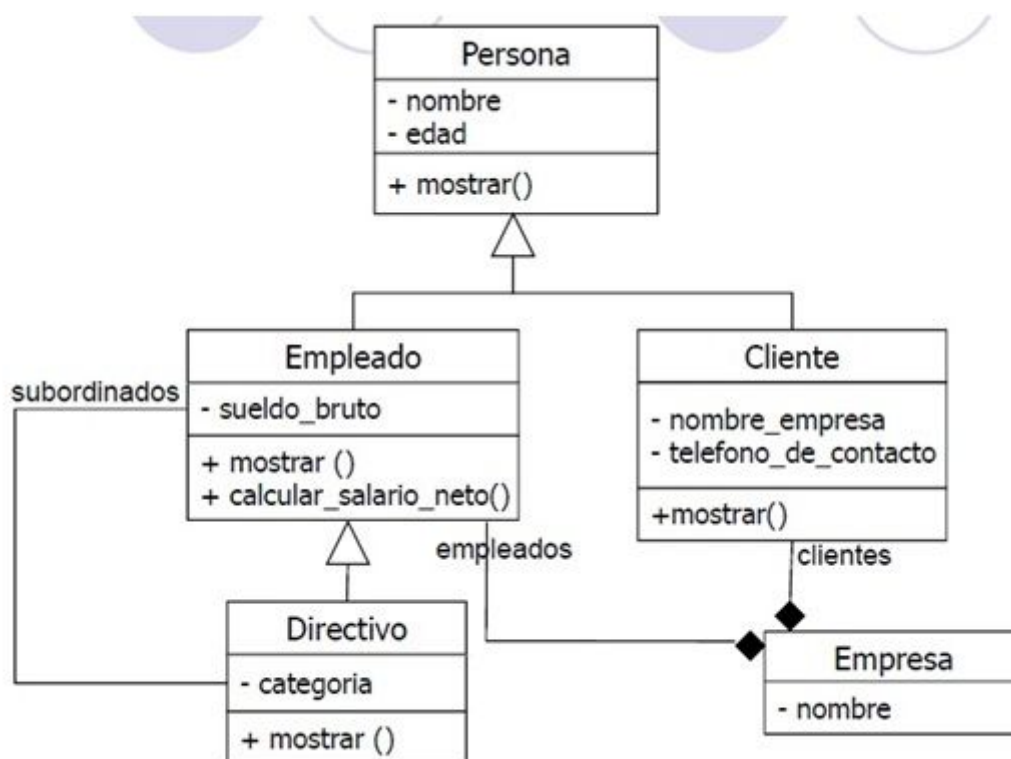
veurem a la UF5), que estan basades en un principi de disseny conegut com [IoC \(Inversion of Control\)](#), on el control del flux del programa el té l'usuari i no pas el propi programa.

[Exemple](#) de nomenclatura de classes segons el patró i ús que se'n fa.

## 6. Herència (extends)


Ens permet definir una classe (filla) com una especialització d'una altra classe existent (mare) amb l'objectiu de reutilitzar codi.

La classe filla hereta (*extens*) les característiques i el comportament de la classe mare, és a dir, les propietats i mètodes, i ens permet afegir de nous.



```
class Empleado extends Persona{
    private double sueldo_bruto;
```

```
    public Empleado (){
```

```
        super(); 
        sueldo_bruto = 0;
```

```
    }
```

```
    public Empleado (String nombre, String apellido1, String apellido2,
        double sueldo_bruto){
```

```

        super(nombre, apellido1, apellido2);
        this.sueldo_bruto = sueldo_bruto;
    }

    public double getSueldo(){
        return sueldo_bruto;
    }

    public void setSueldo(double sueldo_bruto){
        this.sueldo_bruto = sueldo_bruto;
    }
    ...
}

```



- La classe **Object** és la classe mare de totes les classes de JAVA.
- Poden existir tants **nivells** d'herència com necessiten.
- No es permet **l'herència múltiple**, és a dir, una classe filla no pot heretar de dues classes mare diferents.
- Una classe declarada amb la paraula clau **final** no pot tenir classes derivades.
- Una classe declarada amb la paraula clau **abstract** s'utilitza com a "model" per a desenvolupar noves classes derivades. No es poden crear objectes de classes abstract.
  - Una classe abstract pot tenir mètodes abstract, sense codi, però de manera que obligarem a les classes filles a sobreescriure'l.

## **6.1. Reescritura (override) de mètodes**

Podem reescriure mètodes ja existents a la classe mare de dues maneres:

1. Afegint una nova funcionalitat a la ja existent a la classe mare:

```

    public void imprimir(){
        super.imprimir(); //cridem al mètode de la classe mare
        System.out.println("Id: " + id);
        System.out.println("Sueldo: " + Double.toString(sueldo));
    }

```

2. Modificant completament la funcionalitat del mètode de la classe mare:

```

    public boolean equals(Empleado empleado){

```

**//en aquest cas NO cridem al mètode de la classe mare**

```
boolean iguales = false;

if ((this.id != null) && (empleado.getId() != null)){
    if(this.id.equalsIgnoreCase(empleado.getId())){
        iguales = true;
    }
}

return iguales;
}
```



- Els mètodes reescrit substitueixen a tots els efectes als mètodes de la classe mare
- No es poden redefinir mètodes amb la paraula clau **final**

## **6.2. Modificadors d'accés**

Ens permeten modificar la visibilitat de classes, atributs i mètodes i, per tant, el nivell d'encapsulament.

Existeixen 4 nivells d'accés a les classes, atributs i mètodes:

- **private** → només accessible des de l'interior de la classe
- **default** (no posar res) → accessible des de qualsevol classe dins del mateix paquet
- **protected** → des de qualsevol classe dins del mateix paquet o una classe derivada (extends) que estigui fora del paquet
- **public** → des de qualsevol classe

Els nivells d'accés suportats són:

- Classes → default, public, private (classes internes)
- Mètodes → tots
- Atributs → tots

## **6.3. Interfaces**

Una interfície es un conjunt de declaracions de mètodes sense definició (sense codi) que defineixen un "**tipus de conducta**", un comportament "desitjable", la capacitat de "fer o ser" alguna cosa.



Quan una classe implementa (*implements*) una determinada interfície està **obligada** a codificar els mètodes de la interfície. Per tant, és una manera d'assegurar que es defineix el comportament completament.

Quina és la diferència entre **interfaces** i **classes abstractes**?

- Les classes abstractes s'utilitzen com a "model" d'altres classes mentre que les interfaces s'utilitzen per "permetre" una determinada funcionalitat. No estem obligats a reescriure els mètodes de les classes abstractes (si no són mètodes abstractes) però sí el de les interfaces (perquè són buits).
- Les interfaces permeten herència múltiple, és a dir, una classe pot implementar diferents interfaces. L'explicació és que, des del punt de vista funcional, els objectes d'una classe es poden "comportar" de diferents maneres.

Al API de JAVA existeixen moltes interfaces, algunes de les quals utilitzarem, i que ens poden servir com exemple del que significa el concepte d'interfície:

- **Serializable**: Li afegeix als objectes d'una classe la capacitat de ser serialitzats/deserialitzats
- **Comparable**: Li afegeix als objectes d'una classe la capacitat de ser comparats amb el mètode compareTo
- Etc

Per crear una nova interface, i, per tant, definir un nou "comportament":

```
public interface Imprimible{  
    public void imprimir();  
}
```

Per utilitzar una o varies interfaces a una classe:

```
public class ProducteAbstract implements Imprimible,  
Comparable<Object>{  
    ...  
    //estem obligats a definir el mètode imprimir() de la  
    //interface Imprimible  
    public void imprimir(){  
        ...  
    }  
    //estem obligats a definir el mètode compareTo() de la  
    //interface java.lang.Comparable  
    public int compareTo(Object producte){  
        ...  
    }  
}
```

```
}
```

## 6.4. Polimorfisme

Capacitat que una referència d'un objecte es pugui comportar de diferents formes.

A la pràctica, és la relació entre el mètode que es crida i el codi que s'acaba executant (**vinculació** o *binding*) quan tenim herència o interfaces que unifiquin un comportament. Ens permetrà estalviar i reutilitzar codi.

Per exemple, com ja hem fet a la pràctica, podem definir un mètode a la classe ProductesDAO per imprimir tots els productes, tant productes normals com packs, gràcies a un objecte que fa referència a la classe **ProducteAbstract**:

```
public void mostrarTots(){
    for (ProducteAbstract producte : productes.values()) {
        //polimorfisme classes
        producte.imprimir();
    }
}
```

O, per exemple, podem definir un mètode a la classe del **main()** per imprimir qualsevol llista d'objectes (Productes, Clients, Proveïdors, etc) utilitzant la interface **Imprimible** que hem vist a l'apartat anterior:

```
public class IniciVistaControllador{
    ...
    public static void imprimirLlista(HashMap<?, ?> llista){
        //polimorfisme interfaces
        for (Object obj : llista.values()) {
            Imprimible imp = (Imprimible)obj;
            imp.imprimir();
        }
    }
    ...
}
```



- Vinculació temprana (quan escrivim el codi): només podem accedir als mètodes de la classe als que pertany l'objecte concret, és a dir:
  - al fer "*nom\_objecte.*" al IDE, només podem escollir (vincular) els mètodes de la classe a la que pertany l'objecte
  - però podem utilitzar *casting* per canviar un objecte i accedir als seus mètodes de forma "temprana"
- Vinculació tardana (quan executem el codi): executem els mètodes del objecte concret.
  - Recordeu que si reescribim un mètode a una classe filla, el nou mètode substitueix a tots els efectes al de la classe mare i JAVA executarà aquest, el de més baix nivell dintre de la jerarquia de la herència.

## 7. Classes genèriques

A vegades necessitem escriure codi que es pugui utilitzar amb objectes de diferents classes.

Aquí tenim una classe que identifica una cel·la d'una taula, que només pot tenir dintre informació de tipus **String**:

```
public class Cell {
    private String data;

    public void set(String celldata)
    {
        data = celldata;
    }
    public String get() {
        return data;
    }
}
```

La utilitzaríem com es mostra al següent exemple:

```
public class CellDriver {
    public static void main(String[] args) {
        Cell cell = new Cell();
        cell.set("Test");
        System.out.println(cell.get());
    }
}
```

Pero si volem que pugui tenir qualsevol tipus d'objecte (utilitzant polimorfisme de classes) hauriem de definir-la així:

```
public class Cell {
    private Object data;

    public void set(Object celldata)
    {
        data = celldata;
    }
    public Object get() {
        return data;
    }
}
```

Ara també podríem guardar informació de tipus **int**:

```
public class CellDriver {
    public static void main(String[] args) {
        Cell cell = new Cell();
        cell.set(1);
        int num = (int)cell.get();
        System.out.println(num);
    }
}
```

En aquest cas, el problema és que si guardem un String, al fer el get ens donarà un error de **ClassCastException** perquè fem un cast a int.

Una solució bastant farragosa podria ser fer un **instanceof** i preguntar de quin tipus és la dada que hi ha dintre de la cel·la.

La solució idònia és utilitzar les **classes genèriques** fent el següent:

```
public class Cell<T> {
    private T t;

    public void set(T celldata)
    {
        t = celldata;
    }
    public T get() {
        return t;
    }
}
```

Ara ja podem utilitzar aquesta classe per guardar qualsevol tipus de dada, de la següent manera:

```

public class CellDriver {
    public static void main(String[] args) {
        Cell<Integer> integerCell = new Cell<Integer>();
        Cell<String> stringCell = new Cell<String>();
        integerCell.set(1);
        stringCell.set("Test");
        int num = integerCell.get();
        String str = stringCell.get();
    }
}

```



- Com podeu veure, s'informa el tipus de dada entre els signes "<" ">".
- Per consens, les lletres utilitzades per definir una classe genèrica són les següents:
  - E - element (utilitzat a Collections)
  - K - key (utilitzat a Collections)
  - V - value (utilitzat a Collections)
  - N - Number
  - T - Type

## 7.1. Mètodes genèrics

Utilitzant classes genèriques, és habitual trobar-se amb la necessitat d'escriure mètodes genèrics, és a dir, que facin una determinada funcionalitat independentment del tipus de dada de que hi ha dintre de la classe genèrica. Per exemple podem necessitar recorre i imprimir els elements de qualsevol tipus d'una llista, fent el següent:

```

public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.println(elem);
    System.out.println();
}

```



- El println executa el mètode **toString()** de la classe a la que pertany l'element.
- Hem d'utilitzar "?" per indicar a JAVA que el mètode rep una llista de qualsevol tipus com a paràmetre.

## 8. Bibliografía i Webgrafía

- *Introducción al Diseño con Patrones*, Fernando Bellas Permuy.  
<http://www.tic.udc.es/~fbellas/teaching/pfc3/IntroPatrones.pdf>
- *Software Architecture Patterns*, Mark Richards.  
<https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html>
- *Patrones de arquitectura y diseño*  
[https://www.ecured.cu/Patrones\\_de\\_dise%C3%B1o\\_y\\_arquitectura](https://www.ecured.cu/Patrones_de_dise%C3%B1o_y_arquitectura)
- *Patrones de arquitectura vs. Patrones de diseño*, Arleth Paredes Niz.  
<https://arlethparedes.wordpress.com/2012/08/27/patrones-de-arquitectura-vs-patrones-de-diseno/>
- *Reuso del desarrollo a nivel arquitectural - Patrones de diseño - Ejemplo Puerta garaje*  
[https://www.ctr.unican.es/asignaturas/procodis\\_3\\_ii/Doc/patrones\\_de\\_dise%C3%B1o.pdf](https://www.ctr.unican.es/asignaturas/procodis_3_ii/Doc/patrones_de_dise%C3%B1o.pdf)
- *Patrones de diseño de software*, Antonio Leiva  
<https://devexperto.com/patrones-de-diseno-software/>
- Gamma, Erich, Richard Helm Ralph Johnson and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2. [Descàrrega](#)
- *Ejemplos patrones de diseño anteriores en JAVA*  
<https://github.com/LuisBurgos/design-patterns>
- *Exemples patrons de disseny de la Wikipedia*  
[https://es.wikipedia.org/wiki/Categor%C3%ADa:Patrones\\_de\\_dise%C3%B1o](https://es.wikipedia.org/wiki/Categor%C3%ADa:Patrones_de_dise%C3%B1o)
- *Patrones de diseño para Persistencia y Transferencia*  
<https://slideplayer.es/slide/6365814/>
- *Patrones de diseño persistencia datos*, Cecilio Álvarez.  
<http://www.genbetadev.com/java-j2ee/patrones-de-diseno-active-record-vs-dao>
- *Microservicios: Arquitectura de software y Frameworks de código abierto*  
<https://blog.desdelinux.net/microservicios-arquitectura-software-frameworks-codigo-abierto/>
- *Diseño de una aplicación orientada a microservicios*  
<https://docs.microsoft.com/es-es/dotnet/architecture/microservices/multi-container-microservice-net-applications/microservice-application-design>
- *Difference between DTO, VO, POJO, JavaBeans?*  
<https://stackoverflow.com/questions/1612334/difference-between-dto-vo-pojo-javabeans>
- *Polimorfismo en JAVA*



[http://www.buscaminegocio.com/cursos-de-java/polimorfismo-en-java.h  
tml](http://www.buscaminegocio.com/cursos-de-java/polimorfismo-en-java.html)