Tema 3: Disseny

- Un bon disseny de software permet
  1. Manegar de forma viable la complexitat del problema
  2. Desenvolupar el software ràpid
  3. Incloure canvis fàcilment
- Patrons arquitectònics - disseny a gran escala
  * MVC, ECS, SOA
- Patrons de disseny - disseny d'objectes i frameworks a petita i mitjana escala
- Patrons d'estils - solucions a baix nivell orientades a la implementació i al llenguatge.
- Diagrames d'interacció - conjunt d'objectes i les seves relacions
  ↳ De seqüència - ordre temporal dels missatges
  ↳ De col·laboració - organització estructural dels objectes
- Diagrama de classes de disseny - il·lustra les especificacions per classes i interfícies d'una aplicació
- Aspectes propis d'un mal disseny
  1. Rigidesa - cada canvi produeix molts canvis
  2. Immobilitat - no es pot reutilitzar parts del codi
  3. Fragilitat - cada canvi trenca el programa per llocs sense relació conceptual
  4. Viscositat - no es pot canviar el codi sense canviar el disseny
- Aspectes propis d'un bon disseny
  1. Baix acoplament - poca dependència d'altres classes
  2. Bona cohesió - repartiment de responsabilitats, pocs mètodes
- Disseny no orientat a objectes - dependència de codi té el mateix sentit que la dependència d'execució
- Disseny orientat a objectes - l'arquitectura d'execució no restringeix l'arquitectura del codi.

$\pi : G \to G/S_{nn}$

Bijecció entre $S_{nn} \subset H$ i $\bar{H}$

$\pi^{-1}(\bar{H}) = H \Rightarrow H/S_{nn} \cong \bar{H} \Rightarrow |H| = |S_{nn}| \cdot |\bar{H}|$

$\pi : (G \times X) \to X$

Disseny de software

- Tipus de patrons
  1. D'arquitectura - usats a gran escala (Capes)
  2. De disseny - usats en objectes i frameworks de petita o mitjana escala (Facade)
  3. D'estil - solucions de baix nivell orientades a la implementació o al llenguatge (Singleton)
- Test Driven Development (TDD) - basat en unes senzilles regles:
  1. Escriu el nou codi només si el test automàtic ha fallat
  2. Elimina la duplicació de codi.
- Diagrama d'interacció - conjunt d'objectes i les seves relacions, modela el comportament d'un sistema.
  1. Seqüència - ordre temporal dels missatges
  2. Col·laboració - organització estructural dels objectes
- Diagrama de classes de Disseny - el fustra les especialitzacions per classes software i interfícies en una aplicació.
- Un mal disseny es denota per:
  1. Rigidesa - l'impacte d'un canvi al software és imprevisible
  2. Immobilitat - no es pot reutilitzar codi
  3. Fragilitat - a cada canvi, el programa es trenca per un lloc sense relació conceptual
  4. Viscositat - impossibilitat de canviar el codi sense canviar el disseny
- Un bon disseny es denota per:
  1. Baix acoblament - poca connexió, coneixement i dependència d'una classe respecte d'altres
  2. Alta cohesió - poca responsabilitat i mètodes per classe, col·laboració amb altres classes

- En un disseny NO orientat a objectes, el rentit de les dependències en compilació i execució és el mateix.
- En un disseny orientat a objectes, l'arquitectura d'execució no restringeix l'arquitectura del codi.
- Principis de disseny SOLID
  * Single Responsibility Principle - cada classe ha de tenir una única responsabilitat. Permet alta cohesió i entre classes fràgils.
  * Open Closed Principle - els mòduls han de ser oberts per extensió i tancats per modificació. El polimorfisme afavoreix aquest principi.
  * Liskov Substitution Principle - el comportament del mètode d'una classe base no ha de canviar si s'usa amb un objecte d'una classe derivada. Deriva del Open Closed Principle.
  * Interface Segregation Principle - no hem de tenir classes que implementin mètodes que no han d'usar. Permet alta cohesió i entre classes fràgils. Deriva del Single Responsibility Principle
  * Dependency Inversion Principle - les classes d'alt nivell no han de canviar perquè hi hagin les de baix nivell. Les dues haurien de dependre d'abstraccions. Permet un baix acoblament.
- Patrons de disseny - permet solucionar diversos problemes amb un procediment
- Patrons arquitectònics - usats per definir l'estructura lògica del sistema
  1. Patró per capes - un sistema gran que requereix ser descomposat en grups de tasques (components) tals que cada grup de tasques estigi en un nivell determinat d'abstracció (capa). Són conceptuals.
  2. SOA - agrupar serveis als components de l'aplicació mitjançant protocols de comunicació en xarxa.

3. Model - Vista - Controlador - divideix l'aplicació en tres parts interconnectades

* La vista captura l'input de l'usuari i mostra l'estat del model
* El controlador interpreta els events de la vista. Fa crides al model i a la vista
* El model canvia el seu estat (dades) i ho notifica al controlador

- Un sistema orientat a objectes es divideix en tres capes arquitectòniques

1. Presentació - interfície gràfica i gestors (vista)
2. Lògica d'aplicació i objectes del domini - el controlador s'encarrega de processar les peticions de la capa de presentació i el Model s'encarrega dels objectes que representen conceptes del domini.
3. Serveis tècnics - objectes de propòsit general i subsistemes que proveeixen de suport tècnic.

4. DAO - utilitzat per separar l'accés a dades o recursos a baix nivell des de les funcionalitats.

- Patrons de disseny generals (GRASP) - descripció dels principis bàsics d'assignació de responsabilitats a les classes expressades com a patrons.

1. Expert - assignar la responsabilitat a la classe que té la informació per fer aquella tasca. Cada objecte és responsable de mantenir la seva pròpia informació (encapsulament). Permet alta cohesió i baix acoblament.
2. Creador - una classe té la responsabilitat de crear-ne una altra si en conté objectes, en manté un registre de les instàncies... Permet un baix acoblament.

3. Controller - classe encarregada de rebre un event i en dirigir el tractament. No és un objecte de la interfície i fa ús de l'scanner per a demanar coses al Model.

4. Facade - simplifica l'accés a un conjunt de classes proporcionant una única classe que totes utilitzen per comunicar-se amb el conjunt de classes.

- Strategy - podem resoldre un problema amb diferents estratègies. S'usa una interfície comuna per poder encapsular tots els tipus d'algoritmes.
- Singleton - permet que una classe té una sola instància i proporciona un punt d'accés global a ella. (*)
- Factory Method - defineix una interfície per crear objectes, però deixa a les subclasses decidir quina classe ha d'instanciar i consulta el nou objecte creat a través d'una interfície comuna.
- Abstract Factory - defineix una interfície per crear una família d'objectes relacionats sense explícitament especificar les seves classes.
- Observer - permet que un objecte pugui saber els canvis que succeeixen a un altre objecte.
- Composite - defineix jerarquies TOT-PART per tal de permetre als clients tractar objectes simples i compostos de manera uniforme.

(*) Tipus de Singleton

1. Clàssic - bé bo lazy initialization

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

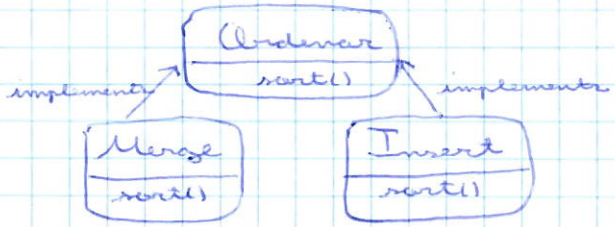
3. Enum - millor opció en Java

```
public enum Singleton {
    // atributs i mètodes
}
```

2. Synchronized - pesado - òptim per multiples threads, Volatile per optimitzar

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

1. Strategy. Tenim un de la interfície Ordenar i volem les classes Merge, Insert,... que la implementen.



En el main sense context tenim això:

```
main() {
Ordenar o;
int[] v = {1, 3, 2}
tipus = llegir String Tipus();
Switch (tipus) {
    case "Merge": o = new Merge();
        break;
    case "Insert": o = new Insert()
        break;
}
o.sort(v);
```

Aquest main vulnera el principi Open-Close, per la qual cosa hem de fer ús del patró Factory Method.

```
public enum FactoryOrdenar {
    INSTANCE,
    public FactoryOrdenar();

    public static Ordenar create Ordenar (String tipus) {
        Ordenar o;
        (*)
        return o;
    }
```

Per ser-ho Singleton

El nou main quedaria així:

```
main() {
    Ordenar o;                          o.sort();
    int[] v = {1, 3, 2};
    o = FactoryOrdenar.create Ordenar ("merge");
```