Génie Logiciel
Patrons logiciels

BAPTISTE PESQUET

Support de cours

https://ensc.gitbook.io/genie-logiciel/

Sommaire

- Introduction
- Patrons d'architecture
- Patrons de conception
  - Chargement tardif
  - Injection de dépendance
  - Singleton
  - Observateur

Introduction

Qu'est-ce qu'un patron logiciel ?

Patron = Motif = *Pattern*

« Bonne pratique standardisée en réponse à un problème »

Solution standard basée sur l'expérience et réutilisable

Aspect technique, indépendant du contexte métier

Conception Vs architecture

Architecture = logiciel considéré de manière globale

- Organisation du logiciel en grandes sous-parties
- Relations entre ces sous-parties

Conception = niveau de granularité plus fin

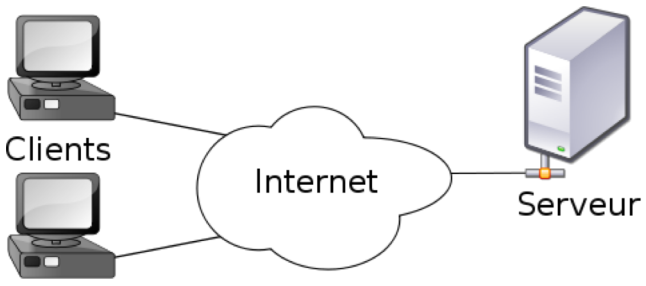
- Plus proche du code source

Distinction non universelle

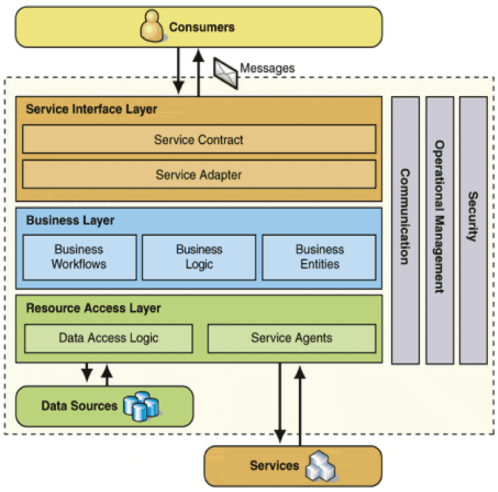
Dépend de la taille du projet

Patrons d’architecture

Architecture client/serveur

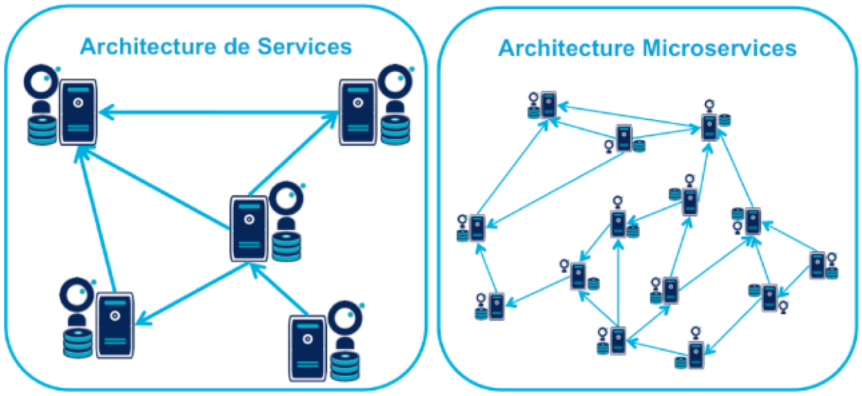


Architecture en couches



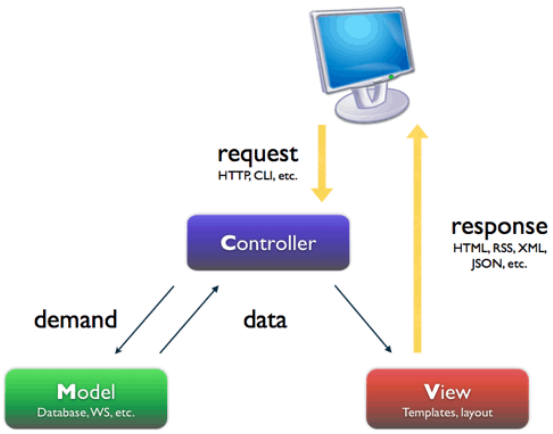
9 sur 28

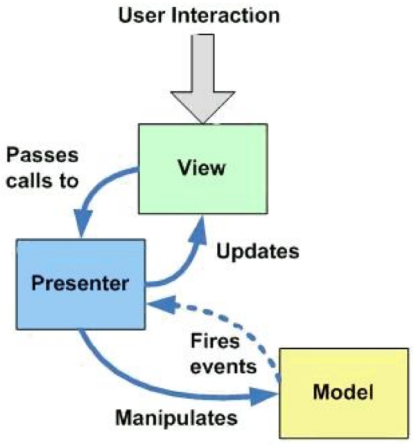
Architecture orientée services (SOA)



10 sur 28

Architectures MVC et MVP





11 sur 28

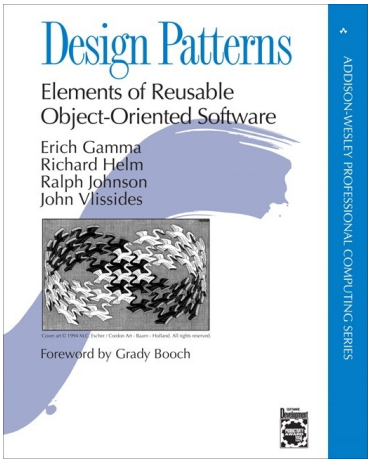
Patrons de conception

Historique

Livre « Design Patterns » du Gang of Four (GoF) publié en 1994

Préfacé par l'un des pères d'UML

Définit et popularise la notion de patron de conception



Les patrons du GoF

23 patrons divisé en 3 familles

- **Creational** : création d'objets
  - *Abstract Factory, Builder, Factory, Prototype, Singleton*
- **Structural** : relations entre objets
  - *Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy*
- **Behavioral** : collaboration entre objets
  - *Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor*

D'autres patrons de conception

Exemples : *Lazy Loading, Dependency Injection*, *Object Pool,* etc

Quelques références :

- Martin Fowler, « Patterns of Enterprise Application Architecture » (PoEAA), 2002
- Steve McConnell, « Code Complete », 2004
- https://en.wikipedia.org/wiki/Software_design_pattern

Chargement tardif

(*LAZY LOADING*)

Objectif

Eviter les initialisations inutiles coûteuses en temps d'exécution et/ou en bande passante

=> Retarder l'accès à une ressource jusqu'au moment de sa première utilisation

Solution

```
// Sans
public class Repository {
  private MySqlConnection connection;
  public Repository() {
    // La connexion est systématiquement faite
    connection = MySqlConnection(…);
    connexion.Open();
  }
  public MySqlConnection Connection {
    get {
      return connection;
    }
  }
```

```
// Avec
public class Repository {
  private MySqlConnection connection;
  public Repository() { }
  public MySqlConnection Connection {
    get {
      // La connexion BD est faite au premier accès
      if (connection == null) {
        connection = MySqlConnection(…);
        connexion.Open();
      }
      return connection;
    }
  }
```

Injection de dépendance

(*DEPENDENCY INJECTION*)

Objectif

Limiter le couplage entre classes

=> Les dépendances entre classes ne sont plus déclarées statiquement mais *injectées* dynamiquement

Souvent couplé à l'utilisation d'interfaces à la place des instances de classe

Solution

```
// Sans
public class Repository { … }

public class App {
  private Repository repository;
  public App() {
    // Dépendance statique entre App et Repository
    repository = new Repository();
  }
```

```
// Avec
public interface IRepository { … }

public class Repository : IRepository { … }

public class App {
  private IRepository repository;
  public App(IRepository repository) {
    // Dépendance injectée à la construction de App
    this.repository = repository;
  }
```


Singleton

Objectif

Limiter le nombre d'instances d'une classe à 1

Fournir un accès global à cette instance

=> Encapsulation du constructeur

Solution

```
// Sans
public class Config {
  public Config() { … }
  public void Action() { … }
}

Config config = new Config();
config.Action();
```

```
// Avec
public class Config {
  private static Config instance = null;
  private Config() { … }
  public static Config Instance {
    get {
      if (instance == null)
        instance = new Config();
      return instance;
    }
  }
  public void Action() { … }
}
Config.Instance.Action();
```

Le singleton, un antipattern ?

Antipattern = !pattern = exemple de mauvaise pratique

Inconvénients du singleton :

- Bloque le nombre d'instances d'une classe à 1
- Pas substituable pour les tests
- Compliqué dans un contexte multithread

=> A utiliser avec modération

Observateur

(*OBSERVER*)

Objectif

Notifier une liste d'objets lors du changement d'un objet cible (sujet)

Exemples de cas d'utilisation :

- Affichages multiples d'une même donnée
- Notifications sur les réseaux sociaux

Solution