Le modèle MVC

Rachid Kadouche
420-KEJ LG

Design Patterns

- Les *design patterns* (DP) sont des modèles de solution à des problèmes fréquemment rencontrés lors des différentes étapes du développement d'applications.
- Ils sont souvent implantés sous forme de classes (dans les langages orientés-objets). Ces solutions se veulent facilement adaptables (donc réutilisables).

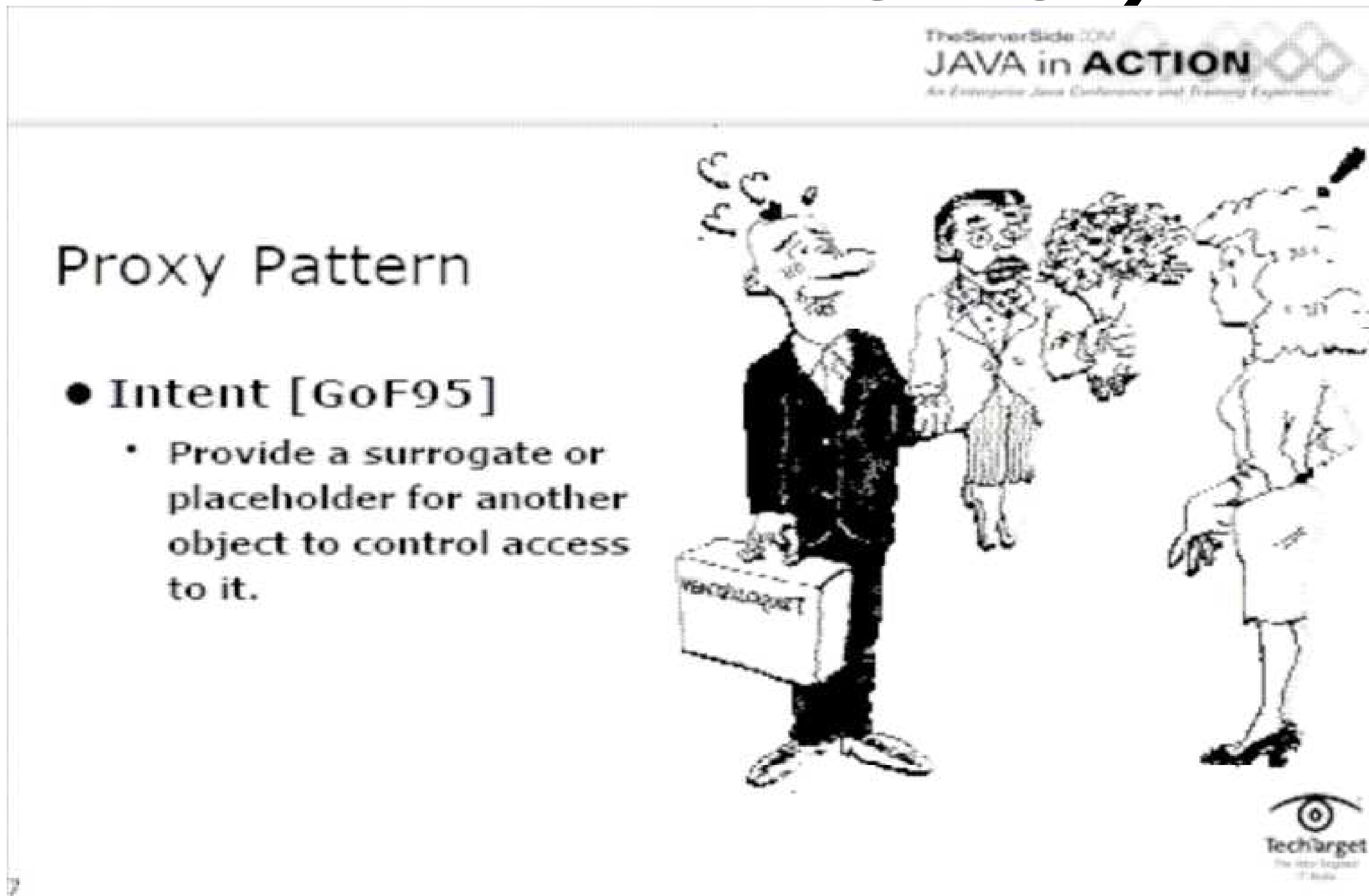
Description

- Les patrons servent à documenter des bonnes pratiques basées sur l'expérience. Ils sont le résultat d'une synthèse de l'expérience acquise par les ingénieurs.

- Les patrons apportent un vocabulaire commun entre l'architecte et le programmeur: Si le programmeur connait le patron de conception MVC, alors l'architecte n'aura pas besoin de lui donner de longues explications et le dialogue se limitera à « ici j'ai utilisé un MVC».

- En programmation informatique, les patrons de conception peuvent être utilisés avant, pendant, ou après le travail de programmation:
  - Utilisé avant, le programmeur utilisera le patron comme guide lors de l'écriture du code source.
  - Utilisé après il servira comme exemple pour relier différents modules de code source déjà écrits, ce qui implique d'écrire le code source nécessaire à leur liaison, et le code qui les fera correspondre au patron de conception.
  - Utilisé pendant le travail de programmation, le programmeur constatera que le code qui vient d'être écrit a des points communs avec un patron existant et effectuera les modifications nécessaires pour que le code corresponde au patron

Exemple de Design Patterns
Le Proxy

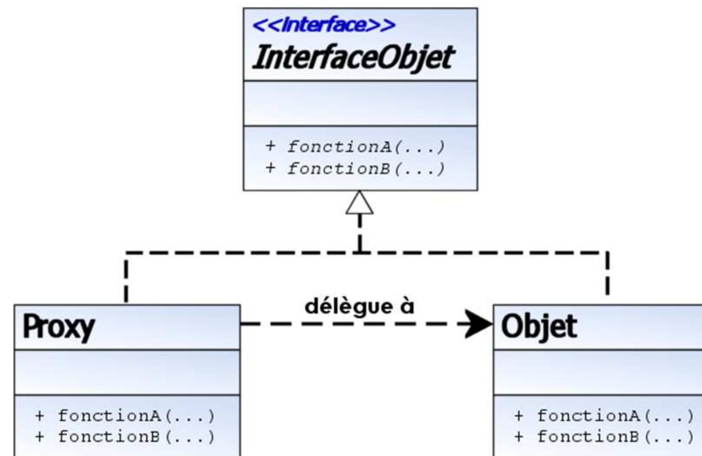


Source: WikiPedia

```java
interface Image {
    public void displayImage();
}

//on System A
class RealImage implements Image {

    private String filename = null;
    /**
     * Constructor
     * @param FILENAME
     */
    public RealImage(final String FILENAME) {
        filename = FILENAME;
        loadImageFromDisk();
    }

    /**
     * Loads the image from the disk
     */
    private void loadImageFromDisk() {
        System.out.println("Loading   " + filename);
    }

    /**
     * Displays the image
     */
    public void displayImage() {
        System.out.println("Displaying " + filename);
    }

}

//on System B
class ProxyImage implements Image {

    private RealImage image = null;
    private String filename = null;
    /**
     * Constructor
     * @param FILENAME
     */
    public ProxyImage(final String FILENAME) {
        filename = FILENAME;
    }

    /**
     * Displays the image
     */
    public void displayImage() {
        if (image == null) {
           image = new RealImage(filename);
        }
        image.displayImage();
    }

}

class ProxyExample {

   /**
    * Test method
    */
   public static void main(String[] args) {
        final Image IMAGE1 = new ProxyImage("HiRes_10MB_Photo1
        final Image IMAGE2 = new ProxyImage("HiRes_10MB_Photo2

        IMAGE1.displayImage(); // loading necessary
        IMAGE1.displayImage(); // loading unnecessary
        IMAGE2.displayImage(); // loading necessary
        IMAGE2.displayImage(); // loading unnecessary
        IMAGE1.displayImage(); // loading unnecessary
    }

}
```

Exemple de Design Patterns

Le Proxy



Exécution du programme

Loading HiRes_10MB_Photo1
Displaying HiRes_10MB_Photo1
Displaying HiRes_10MB_Photo1
Loading HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo1

Source: WikiPedia

Design Patterns
Avantages et inconvénients

Avantages

- Les *design patterns* ont une architecture facilement compréhensible et identifiables pour un programmeur, ce qui permet aux autres programmeurs de s'y retrouver plus rapidement.
- Ils permettent de ne pas réinventer la roue.
- Et surtout, ils nous offrent des modèles de solution à des problèmes de «design» d'application fréquents.

Inconvénients

- Les *design patterns* ne sont pas une solution toute faite, ce sont plutôt des méthodes de résolutions.
- L'utilisation des *design patterns* ne fait pas l'unanimité en entreprise, loin de là. La plupart des entreprises ne s'en occupent pas du tout. D'autres en font une religion. Et certains ne les utilisent que lorsqu'ils répondent à un besoin (c'est peut-être la façon la plus sage...).

MVC

- L'architecture **Modèle Vue Contrôleur** (MVC *Model View Controler*) est l'un des *design patterns* les plus pratiques. Il est demeuré actuel car il permet de «standardiser» certains aspects des architectures d'applications modernes. Il a de plus été repris par plusieurs outils de développement et il est la base de plusieurs type d'applications client-serveur.

Buts de MVC

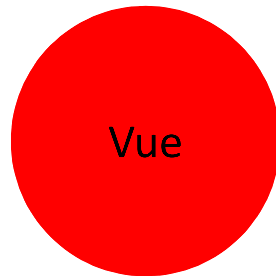
- L ’architecture Model-View-Controller a pour objectif d ’organiser une application interactive en séparant :
  - les données
  - la représentation des données
  - le comportement de l ’application

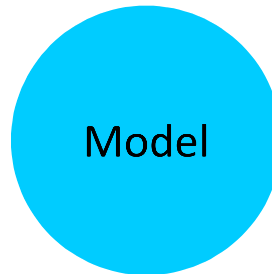
Structure de MVC

- Le modèle représente la structure des données dans l ’application et les opérations spécifiques sur ces données.
- Une Vue présente les données sous une certaine forme à l ’utilisateur, suivant un contexte d ’exploitation.
- Un Contrôleur traduit les interactions utilisateur par des appels de méthodes (comportement) sur le modèle et sélectionne la vue appropriée basée sur l’état du modèle.

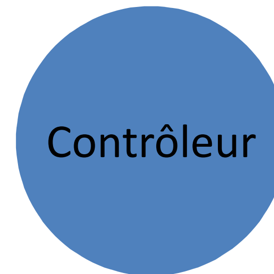
Vue globale MVC

Vue présentent l'information à l'utilisateur

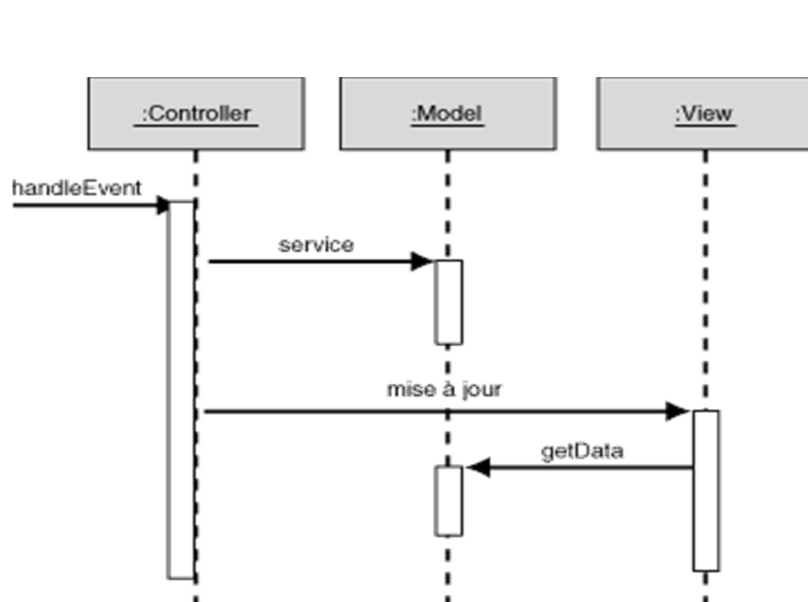


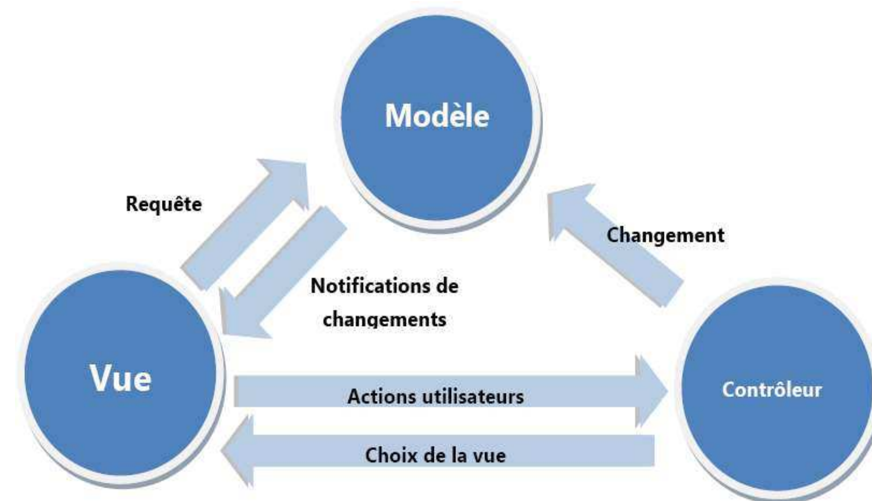


Modèle implémentent les fonctionnalités de l'application



Contrôleurs gèrent les entrées de l'utilisateur





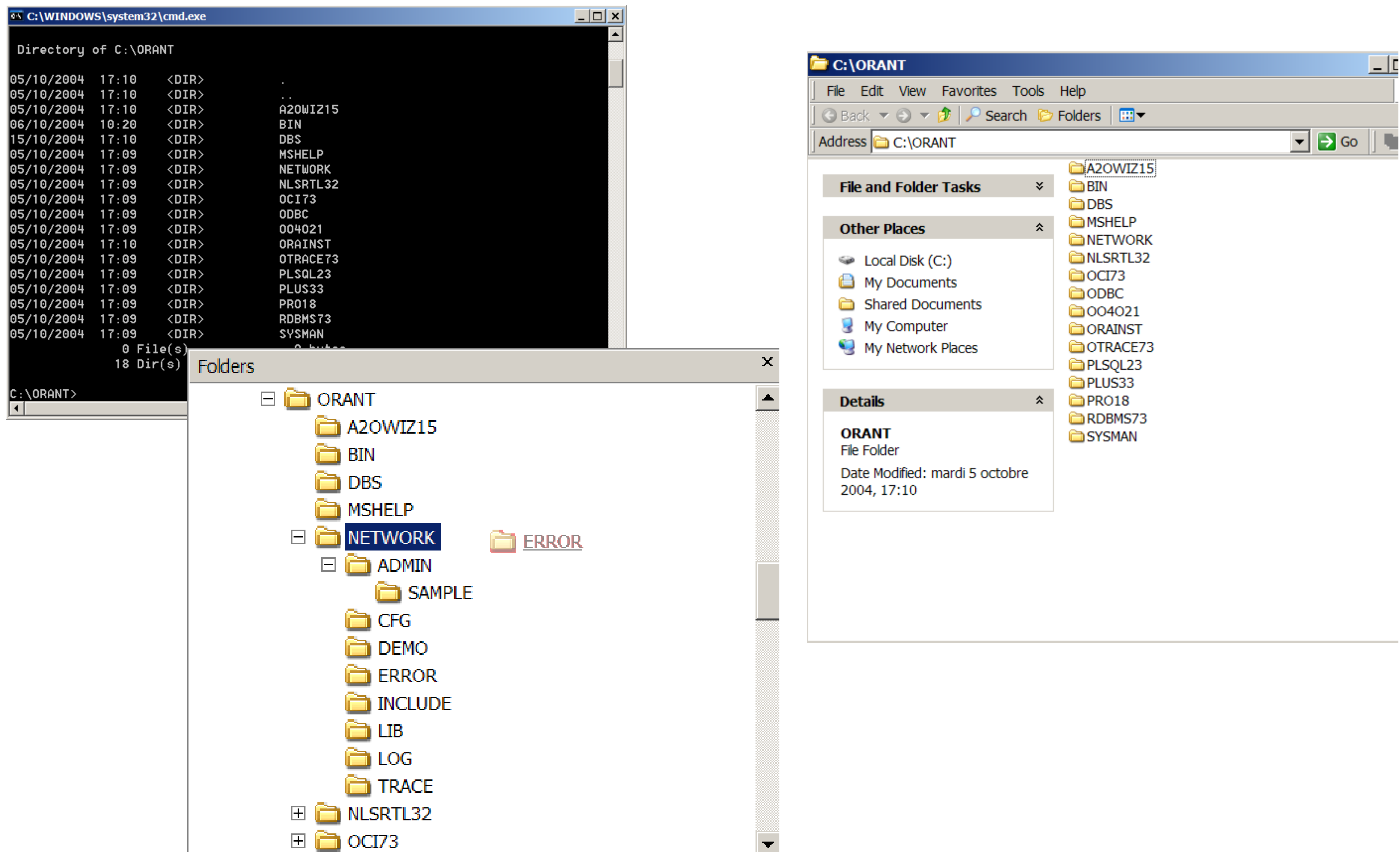
Indépendance des modules

- MVC conserve un modèle indépendant des vues.
- Le modèle ne nécessite jamais d'informations spécifiques sur les vues.
- Si le modèle devait notifier les changements aux vues, il fournit un mécanisme pour alerter les autres objets des changements d'état sans pour autant introduire de dépendances entre eux.
- Chaque vue implémente l'interface Observer et s'enregistre auprès du modèle. Le modèle tient à jour la liste de tous les observateurs qui s'abonnent aux changements. Lorsqu'un modèle change, il effectue une itération sur tous les observateurs enregistrés et les notifie du changement. Cette approche est généralement appelée «éditer-s'abonner».

Différence avec l'architecture trois tiers

- Dans l'architecture trois couches:
  - chaque couche communique seulement avec ses couches adjacentes (supérieures et inférieures)
  - le flux de contrôle traverse le système de haut en bas; les couches supérieures contrôlent les couches inférieures
  - les couches supérieures sont toujours sources d'interaction (clients) alors que les couches inférieures ne font que répondre à des requêtes (serveurs).
  - si une vue modifie les données, toutes les vues concernées par la modification doivent être mises à jour.
- Dans le modèle MVC,
  - la vue peut consulter directement le modèle (lecture) sans passer par le contrôleur. Par contre, elle doit nécessairement passer par le contrôleur pour effectuer une modification (écriture).
  - le flux de contrôle est inversé par rapport au modèle en couches, le contrôleur peut envoyer des requêtes à toutes les vues de manière à ce qu'elles se mettent à jour.

Un exemple de multiplicité des Vues



Exemple 1

```cpp
class Button; // Prewritten GUI element

class GraphGUI {
public:
    GraphGUI() {
        _button = new Button("Click Me");
        _model = new GraphData();
        _controller = new GraphController(_model, _button);
    }
    ~GraphGUI() {
        delete _button;
        delete _model;
        delete _controller;
    }

    drawGraph() {
        // Use model's data to draw the graph somehow
    }
    ...

private:
    Button*              _button;
    GraphData*           _model;
    GraphController*     _controller;
};

class GraphData {
public:
    GraphData() {
        _number = 10;
    }
    void increaseNumber() {
        _number += 10;
    }
    const int getNumber() { return _number; }
private:
    int _number;
};

class GraphController {
public:
    GraphController(GraphData* model, Button* button) {
        __model = model;
        __button = button;
        __button->setClickHandler(this, &onButtonClicked);
    }

    void onButtonClicked() {
        __model->increaseNumber();
    }

private:
    // Don't handle memory
    GraphData*    __model;
    Button*       __button;
};
```

Exemple 2

- Une classe de Interface graphique : elle contient la description de l'interface graphique avec ses différents composants ; **c'est la vue**.
- Une classe de l'application elle-même : elle contient tous les attributs et méthodes qui doivent participer à la tâche principale à accomplir : **c'est le modèle**.
- Une classe de contrôle qui contient l'ensemble des listeners et qui, lorsque des événements parviennent via la vue, prévient le modèle en conséquence : **c'est le contrôleur**.