IPRO

INTRODUCTION AUX PATRONS DE CONCEPTION

Guillaume Bouyer

ENSIIE

www.ensiie.fr/~bouyer

PLAN DU COURS

LES PATRONS EN PROGRAMMATION

Les patrons décrivent des solutions générales à des problèmes récurrents

- Aide à la compréhension
- Moyen de communication
- Capitalisation de l'expérience en génie logiciel
- Aide à la construction de logiciels

TYPES DE PATRONS

Design Patterns

Structures de conception

Architectural Patterns

Schémas d'organisation structurelle du logiciel (haut niveau)

Ex : MVC

Idioms* ou *coding patterns

Solutions dans un langage particulier

Anti-patterns

Mauvaises solutions courantes (ou comment en sortir)

Organizational patterns

Schémas d'organisation (humaine) du développement

DESIGN PATTERN

Solution générale à un problème courant en conception logicielle

- Définition d'interactions entre des classes/objets
- Indépendant du langage de programmation et du paradigme :
  - un design pattern s'adapte
  - ne s'applique pas aveuglément, dépend de son contexte
  - peuvent se combiner
  - est différent d'une bibliothèque
- Vise à l'amélioration de l'efficacité, de la robustesse, du temps de développement, de la lisibilité du code

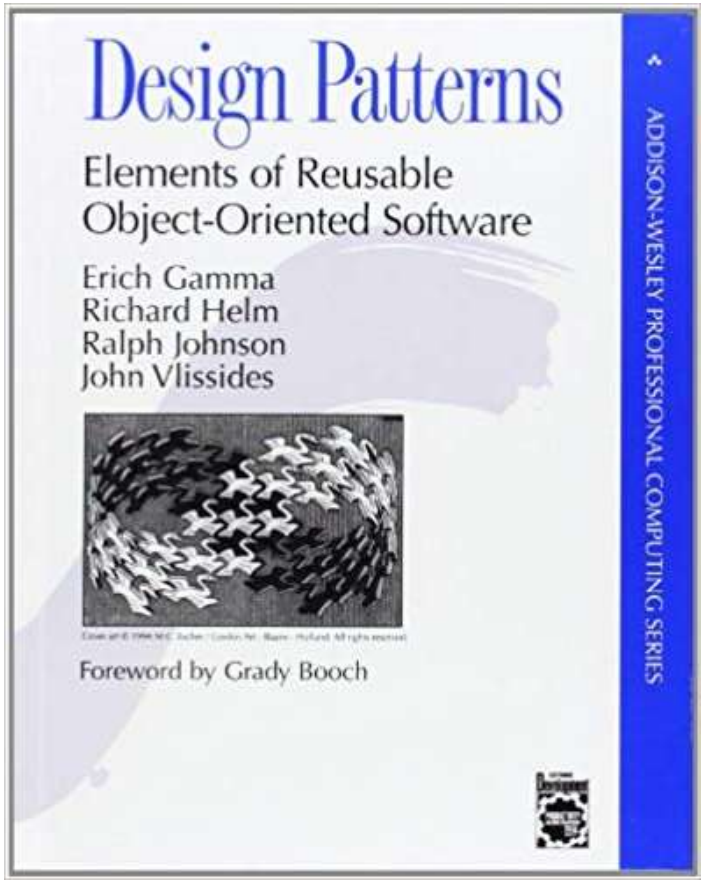
AVANTAGES

- Vocabulaire formalisé
- Capitalisation de l'expérience
- Niveau d'abstraction élevé
- Complexité réduite
- Temps de conception et développement réduit
- Guide/catalogue de solutions
- Aident à respecter les 5 principes SOLID

INCONVÉNIENTS

- Synthèse nécessaire :
  - Comment reconnaître une situation où un pattern s'applique ?
- Apprentissage :
  - Catalogue volumineux : plus de 150 patrons répertoriés
  - Différents niveaux : des patrons de patrons
- Adaptation du patron à son contexte

LE "GANG OF FOUR"



1995

PATRONS DU GOF

Créationnels

Concernent le processus d'instanciation, d'initialisation et de configuration des classes et des objets

Structuraux

Concernent l'organisation des classes et des objets dans le programme, la manière dont ils sont connectés, composés pour obtenir des structures plus complexes

Comportementaux

Concernent les interactions dynamiques des classes et des objets, leurs collaborations

Concernent les algorithmes et la répartition des responsabilités entre les objets

PATRONS DU GOF

Créationnels (5)	**Structurels (7)**	**Comportementaux (11)**
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Façade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

CREATIONAL PATTERNS : PRINCIPES GÉNÉRAUX

Permettent des mécanismes efficaces de création d'objets, ce qui augmente la flexibilité et la réutilisation du code

Permettent dynamiquement ou statiquement de préciser les paramètres de la création : *QUOI* (l'objet), *QUI* (l'acteur), *COMMENT* (la manière) et *QUAND* (le moment)

Notamment les *fabriques* : quand plusieurs classes disponibles mais qu'on ne saura qu'au runtime laquelle et selon certaines conditions on doit instancier

CREATIONAL PATTERNS (1/2)

“Simple factory” (non GoF)

Cache les détails de la création dans une méthode qui retourne l'instance créée selon un paramètre (`switch/case` avec des `new`)

A modifier en cas d'ajouts d'objets

Factory Method

Une interface pour créer un objet, mais des sous-classes choisissent la classe à instancier : une méthode de création abstraite redéfinie dans des classes filles pour chaque type de produits, eux-mêmes hiérarchisés

En plus, permet de contrôler l'algorithme de création et la gestion de l'objet créé

Abstract Factory

Permet d’instancier des familles de produits dépendant les uns des autres sans avoir à préciser leur type concret

Ex : “si le produit est un A, alors c'est le service SA qui le gère ; si c'est un B, alors c'est SB”

CREATIONAL PATTERNS (2/2)

Builder

Pour la création d'objets complexes dont les différentes parties doivent être créées suivant un certain ordre ou algorithme spécifique

Peut remplacer un constructeur avec de nombreux et optionnels paramètres

Prototype

Permet de produire de nouveaux objets en clonant des objets existants sans compromettre leurs caractéristiques internes

Singleton

Permet d'assurer qu'une classe possède une seule instance et en fournir un point d'accès global

STRUCTURAL PATTERNS : PRINCIPES GÉNÉRAUX

Permettent la construction de hiérarchies de classes et des relations entre les différentes classes simples et efficaces, ce qui facilite l'évolution de l'application

STRUCTURAL PATTERNS (1/2)

Adapter

Permet à des objets ayant des interfaces incompatibles de travailler ensemble

Convertit l'interface d'une classe en une autre interface que les clients attendent

Bridge

Permet de diviser une classe massive ou un ensemble de classes étroitement liées en deux hiérarchies distinctes, l'abstraction et l'implémentation, qui peuvent être développées indépendamment l'une de l'autre.

Composite

Compose des objets dans des structures arborescentes et permet aux clients de travailler avec ces structures comme s'il s'agissait d'objets individuels

STRUCTURAL PATTERNS (2/2)

Decorator

Permet d'attacher dynamiquement de nouveaux comportements aux objets en les plaçant à l'intérieur d'objets enveloppe qui contiennent ces comportements
Alternative souple à l'héritage pour étendre les fonctionnalités

Facade

Fournit une interface unifiée et plus simple à un système complexe de classes, bibliothèque ou framework
Définit une interface de haut niveau qui rend le sous-système plus facile à utiliser

Flyweight

Permet de partager les parties communes entre plusieurs objets, et ainsi gagner de l'espace mémoire

Proxy

Fournit un substitut ou un conteneur pour un autre objet afin d'en contrôler l'accès.

BEHAVIOURAL PATTERNS : PRINCIPES GÉNÉRAUX

Permettent la communication, la distribution efficace et sûre des comportements entre les objets

BEHAVIOURAL PATTERNS (1/4)

Chain of Responsibility

Évite de coupler l'expéditeur d'une requête à son destinataire en donnant à plus d'un objet une chance de traiter la demande

Enchaîne les récepteurs et fait passer la requête le long de la chaîne jusqu'à ce qu'un objet la traite.

Command

Permet d'encapsuler une requête en objet indépendant, qui peut être utilisé pour paramétrer des objets avec différentes requêtes, des files d'attente ou des requêtes de log

Permet des opérations annulables

Interpreter

Spécifie comment évaluer les phrases dans un langage

BEHAVIOURAL PATTERNS (2/4)

Iterator

Fournit un moyen d'accéder séquentiellement aux éléments d'un agrégat sans exposer sa représentation sous-jacente

Mediator

Définit un objet qui encapsule comment un ensemble d'objets interagit

Favorise un couplage lâche en empêchant les objets de se référer explicitement, et permet de modifier leur interaction de manière indépendante

Memento

Permet de capturer et externaliser l'état interne d'un objet sans exposer sa structure afin qu'il puisse être restauré ultérieurement

BEHAVIOURAL PATTERNS (3/4)

Observer

Définit une dépendance d'un à plusieurs entre des objets afin que lorsqu'un objet change d'état, tous ses dépendants sont notifiés et mis à jour automatiquement

State

Autorise un objet à modifier son comportement lorsque son état interne change

L'objet semblera changer de classe

Strategy

Définit une famille d'algorithmes, les encapsule chacun et les rend interchangeables

Permet à l'algorithme de varier indépendamment des clients qui l'utilisent

BEHAVIOURAL PATTERNS (4/4)

Template Method

Définit le squelette d'un algorithme dans une opération, reportant certaines étapes vers des sous-classes

Permet aux sous-classes de redéfinir certaines étapes de l'algorithme sans modifier sa structure

Visitor

Permet de définir une nouvelle opération sans modifier les classes des objets sur lesquels elle fonctionne

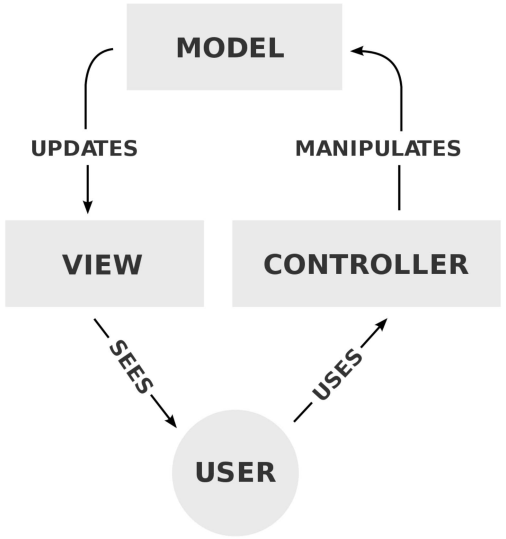
EXEMPLE D'ARCHITECTURAL PATTERN

MODEL-VIEW-CONTROLLER

PRINCIPE MVC

Séparation claire entre données métier, présentation et traitements

A la base de nombreux frameworks, sous des formes diverses (adaptations)



COMPOSANTS

Modèle

- Couche métier
- Gestion des données du programme et de leurs traitement
- Comportement de l'application

Vue

- Interface (graphique le plus souvent) avec l'utilisateur
- Présentation de l'état du modèle
- Gestion des requêtes utilisateurs

Contrôleur

- Gestion des événements
- Modification du modèle sur demande de l'utilisateur

Pour un modèle donné, plusieurs vues et contrôleurs possibles

EXEMPLE JAVA

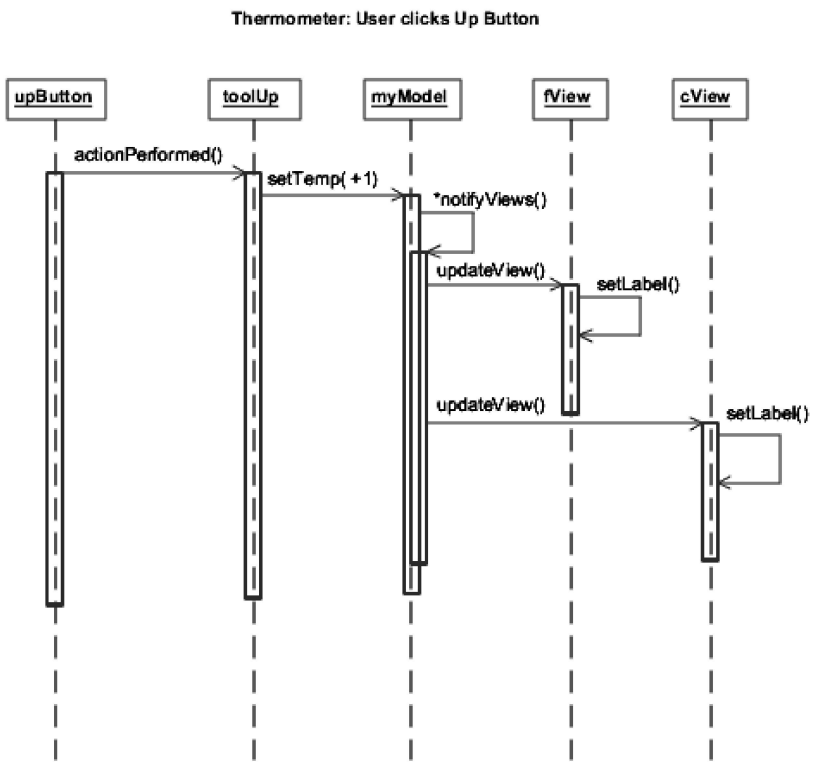
Pattern Observer + librairie graphique `Swing`

- Modèle
  - est observable : `extends java.util.Observable`
  - Après chaque changement d'état, appelle `setChanged ()` et `notifyObservers ()`
- Vue
  - observe le Modèle : `implements java.util.Observer`
  - est ou contient une `java.awt.Frame`
  - contient des panels, boutons, textes...
  - possède une méthode `void update(Observable obs, Object obj)` appelée lors de la notification

EXEMPLE JAVA

- Controleur
  - est un listener : `implements java.awt.event.ActionListener`
  - connaît le modèle et la vue (attributs)
  - peut effectuer une action (`actionPerformed(java.awt.event.ActionEvent e)`) qui agit sur le modèle via ses méthodes
- Un composant principal fait le lien entre M, V et C
  - instancie le modèle, la vue et le contrôleur
  - ajoute la vue comme observer du modèle
  - ajoute le controleur comme listener d'un élément adéquat de la vue

EXEMPLE JAVA



EXERCICES

PROBLÈME : TRAITEMENT D'INFORMATIONS DE SOURCES DIFFÉRENTES

Votre programme doit lire des informations à partir de deux sources de données possibles, par ex. un fichier CSV et une base de données. Il doit ensuite traiter ces données de la bonne manière, et enfin générer des résultats en tant qu'autres fichiers CSV. Trois étapes sont impliquées.

- Lire les données provenant de la source de données correspondante
- Traiter les données selon la source
- Ecrire la sortie dans les fichiers CSV

Proposez une architecture (sous forme de diagramme de classe).

PROBLÈME : OBJET UNIQUE

Proposez une solution qui assure qu'une classe ne sera instanciée qu'une fois (ex. game manager)

PROBLÈME : NOTIFICATIONS

On souhaite proposer une solution aux clients d'une boutique en ligne pour les notifier automatiquement dès qu'un produit qui les intéresse devient disponible. Proposez une architecture.

PROBLÈME : CRÉATION DE POINTS

On souhaite ajouter dans la classe Point2D un 2ème constructeur qui prend 2 coordonnées polaires (double, double), alors qu'on avait déjà Point2D(double x, double y). Est-ce possible ? Proposez une solution

PROBLÈME : CRÉATION D'UNITÉS

On suppose qu'on dispose d'une hiérarchie d'Unités concrètes (Soldier, Healer, Tank...). Dans 2 modules clients (ex. UI et Network) on veut pouvoir instancier à la volée différentes unités en utilisant leur “type”. On ne veut pas avoir à modifier les clients lorsqu'il y aura de nouvelles unités. Proposez une solution.

PROBLÈME : UN PEU D'ACTION

On veut maintenant que nos unités Soldier, Healer, Tank et les prochaines que l'on imaginera puissent effectuer une action d'attaque selon les régles imaginées par les game designers : normal shoot, debuff shoot, arrow shoot, no shoot...

Cette action doit pouvoir changer selon le type d'unité, mais aussi selon les instances d'un même type, voire même selon le contexte pour une instance donnée.

Proposez une solution.