Les Designs Patterns 2

Yassamine Seladji

yassamine.seladji@gmail.com

18 septembre 2017

Les Designs Patterns (Les Modèles de Conceptions)

Les Designs Patterns :

- C'est une technique d'architecture logicielle.
- Modélisation d'une solution à un problème récurent.
- Baser sur une expérience éprouvée de conception.
- Baser sur la conception orientée objet.
- Décomposer en trois parties :
  - Les patterns de création : relier à la création d'objets.
  - Les patterns de structure.
  - Les patterns de comportement.

Les Patterns de Structures

Les Patterns de Structures

- Faciliter la conception en identifiant la meilleur manière de relier les objets et classes entre eux afin de créer des systèmes plus important.

Les Patterns de Structures

- Faciliter la conception en identifiant la meilleur manière de relier les objets et classes entre eux afin de créer des systèmes plus important.
- Montrer comment coller différents morceaux d'un système afin qu'il soit flexible et extensible.

Les Patterns de Structures

- Faciliter la conception en identifiant la meilleur manière de relier les objets et classes entre eux afin de créer des systèmes plus important.
- Montrer comment coller différents morceaux d'un système afin qu'il soit flexible et extensible.
- Utiliser l'héritage pour composer des classes et/ou des interfaces.

Les Patterns de Structures

- Faciliter la conception en identifiant la meilleur manière de relier les objets et classes entre eux afin de créer des systèmes plus important.
- Montrer comment coller différents morceaux d'un système afin qu'il soit flexible et extensible.
- Utiliser l'héritage pour composer des classes et/ou des interfaces.
- Décrire la manière dont les objets doivent être composer entre eux afin réaliser de nouvelles fonctionnalités.

Les Patterns de Structures

- Faciliter la conception en identifiant la meilleur manière de relier les objets et classes entre eux afin de créer des systèmes plus important.
- Montrer comment coller différents morceaux d'un système afin qu'il soit flexible et extensible.
- Utiliser l'héritage pour composer des classes et/ou des interfaces.
- Décrire la manière dont les objets doivent être composer entre eux afin réaliser de nouvelles fonctionnalités.
- La composition d'objets ajoute une plus grande flexibilité, car cette composition peut être changer à l'exécution.

Les Patterns de Structures : Adapter

Nom :

Le Modèle adaptateur (The Adapter Pattern).

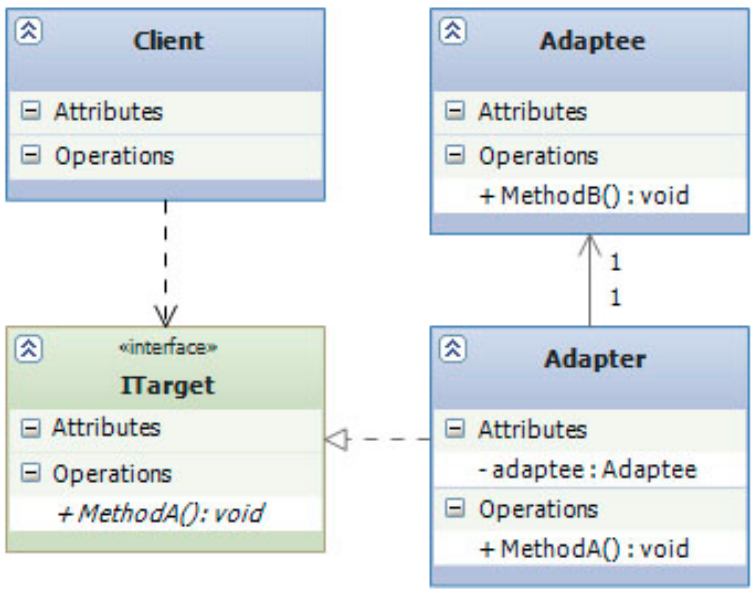
Problème :

On utilise ce design pattern lorsque :

- L'interface d'une classe est incompatible avec l'interface du client.
- Des classes avec des interfaces différents doivent fonctionner ensembles.

Les Patterns de Structures : Adapter

Solution :



Client : représente la classe qui à besoin d'utiliser une interface incompatible.
ITarget : définis l'interface que le client utilise.
Adaptee : représente la classe des fonctions dont le client a besoin.
Adapter : définis la classe qui traduit l'interface de **Adaptee** à l'interface of **Client**.

Les Patterns de Structures : Adapter

Exemple : Adapter une liste doublement chainée en une pile

```
interface Stack {
   void push(Object o);
   Object pop();
   Object top();
}

/* Liste doublement chainée */
class DList {
   public void insert (DNode pos, Object o) { ... }
   public void remove (DNode pos) { ... }
   public void insertHead (Object o) { ... }
   public void insertTail (Object o) { ... }
   public Object removeHead () { ... }
   public Object removeTail () { ... }
   public Object getHead () { ... }
   public Object getTail () { ... }
}
```

```
/* Adapt DList class to Stack interface */
class DListImpStack extends DList implements Stack {
   public void push(Object o) {
      insertTail(o);
   }
   public Object pop() {
      return removeTail();
   }
   public Object top() {
      return getTail();
   }
}
```

- L'interface ITarget $\Longrightarrow$ L'interface Stack.
- La classe Adaptee $\Longrightarrow$ La classe DList.
- La classe Adapter $\Longrightarrow$ La classe DListImpStack.

Les Patterns de Structures : Adapter

Conséquences :

- Avantages : Facile à implémenter, l'ajout d'une seul classe.
- Limitations : Ne fonctionne pas avec les sous classe de la classe **Adaptee**.

Les Patterns de Structures : Composite

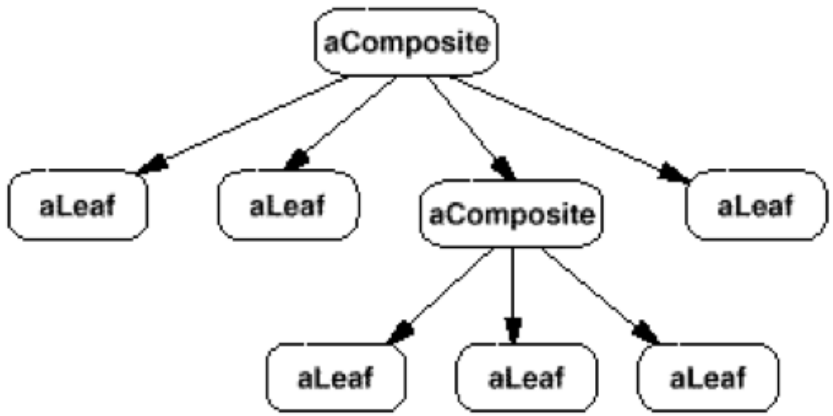
Nom :

Le Modèle de composition (The Composite Pattern).

Problème :

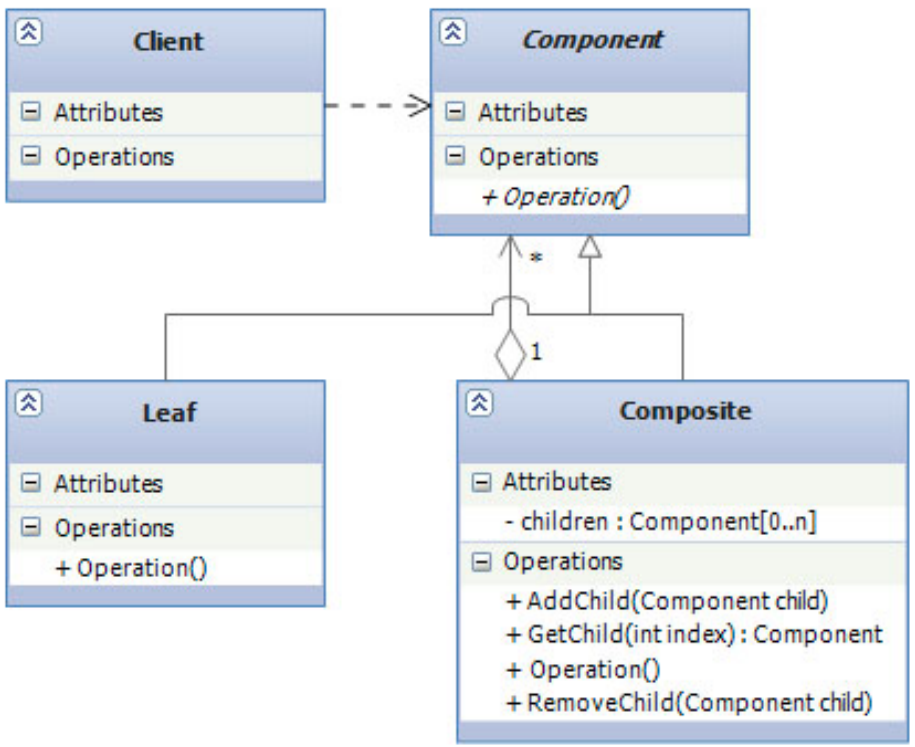
On utilise ce design pattern pour :

- Créer des modèles hiérarchiques d'objets.
- Établir des structures arborescentes entre des objets et les traiter uniformément.
- Masquer aux clients la nature et la complexité des objets qu'il manipule. (objet simple/objet composé).



Les Patterns de Structures : Composite

Solution :



Client : manipule les objets de la classe **Composite** en passant par l'interface **Component**.

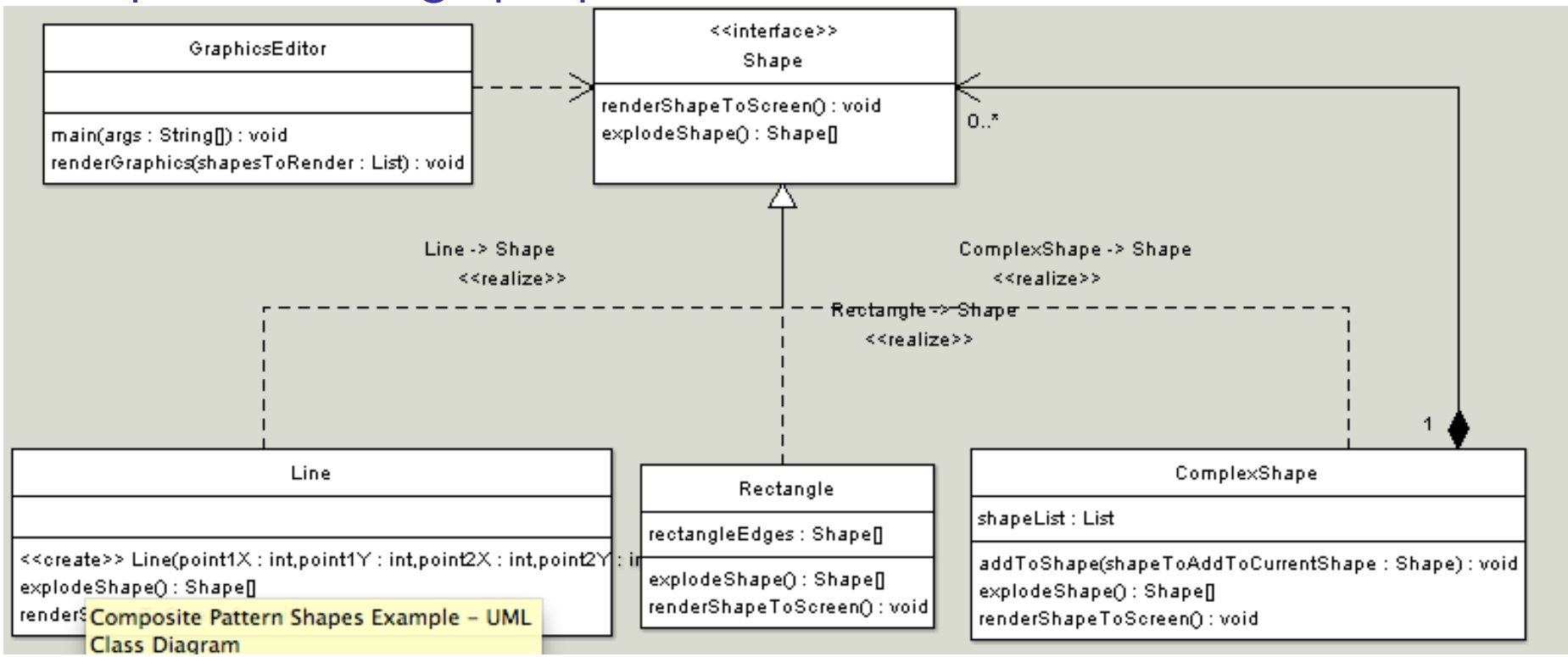
Component : représente l'interface des objets de la composition.

Composite : définis le comportement des composants.

Leaf : représente la classe des objets utilisés comme éléments (children) dans **Composite**.

Les Patterns de Structures : Composite

Exemple : éditeur graphique



Les Patterns de Structures : Composite

Exemple : éditeur graphique

```
public class Line implements Shape {
    public Line(int point1X, int point1Y,
            int point2X, int point2Y) {}

    @Override
    public Shape[] explodeShape() {
        Shape[] shapeParts = {this};
        return shapeParts;
    }

    public void renderShapeToScreen() {
        // Afficher la forme dans l'ecran
    }

}
```

```
public class Rectangle implements Shape{

    Shape[] rectangleEdges = {
            new Line(-1,-1,1,-1),new Line(-1,1,1,1),
            new Line(-1,-1,-1,1),new Line(1,-1,1,1)
            };

    @Override
    public Shape[] explodeShape() {
        return rectangleEdges;
    }

    public void renderShapeToScreen() {
        for(Shape s : rectangleEdges){
            s.renderShapeToScreen();
        }
    }
}
```

Les Patterns de Structures : Composite

Exemple : éditeur graphique

```
public class ComplexShape implements Shape{

    List<Shape> shapeList = new ArrayList<Shape>();

    public void addToShape(Shape shapeToAddToCurrentShape)

        shapeList.add(shapeToAddToCurrentShape);
    }

    public Shape[] explodeShape() {
        return (Shape[]) shapeList.toArray();
    }

    public void renderShapeToScreen() {

        for(Shape s: shapeList){
            s.renderShapeToScreen();

        }
    }
}
```

```
public class GraphicsEditor {

    public static void main(String[] args) {

        List<Shape> allShapesInSoftware = new ArrayList<Shape>();

        Shape lineShape = new Line(0,0,1,1);
        allShapesInSoftware.add(lineShape);

        Shape rectangelShape = new Rectangle();
        allShapesInSoftware.add(rectangelShape);

        ComplexShape complexShape = new ComplexShape();
        complexShape.addToShape(rectangelShape);
        complexShape.addToShape(lineShape);
        allShapesInSoftware.add(complexShape);

        ComplexShape veryComplexShape = new  ComplexShape();
        veryComplexShape.addToShape(complexShape);
        veryComplexShape.addToShape(lineShape);
        allShapesInSoftware.add(veryComplexShape);

        renderGraphics(allShapesInSoftware);
    }

    private static void renderGraphics(List<Shape> shapesToRender){
        for(Shape s : shapesToRender){
            s.renderShapeToScreen();
        }
    }
}
```

Les Patterns de Structures : Composite

Conséquences :

- Avantages :
  - L'ajout de nouveaux composants est simple cela grâce à la hiérarchie des classes.
  - Le client ne se soucis pas de l'objet accédé.
- Limitations : La vérification du type des composants est difficile.

Exemples :

- java.awt.Component
- java.awt.Container

Les Patterns de Comportement

Les Patterns de Comportement

Les patterns de comportement :

- Traitent la partie algorithmique ainsi que la distribution des responsabilités entre objets.
- Décrivent la communication et l'interconnexion entre les classes et les objets.

Les Patterns de Comportement

Deux types de patterns :

- Les patterns de classes : utilisation de l'héritage pour répartir les comportement entre classes.
- Les patterns d'objets : utilisation de l'association entre objets pour décrire :
  - La coopération entre groupes d'objets.
  - Les dépendances entre objets.
  - L'encapsulation d'un comportement dans un objet et la délégation des requêtes à d'autres objets.

Les Patterns de Comportement : Observer

Nom :

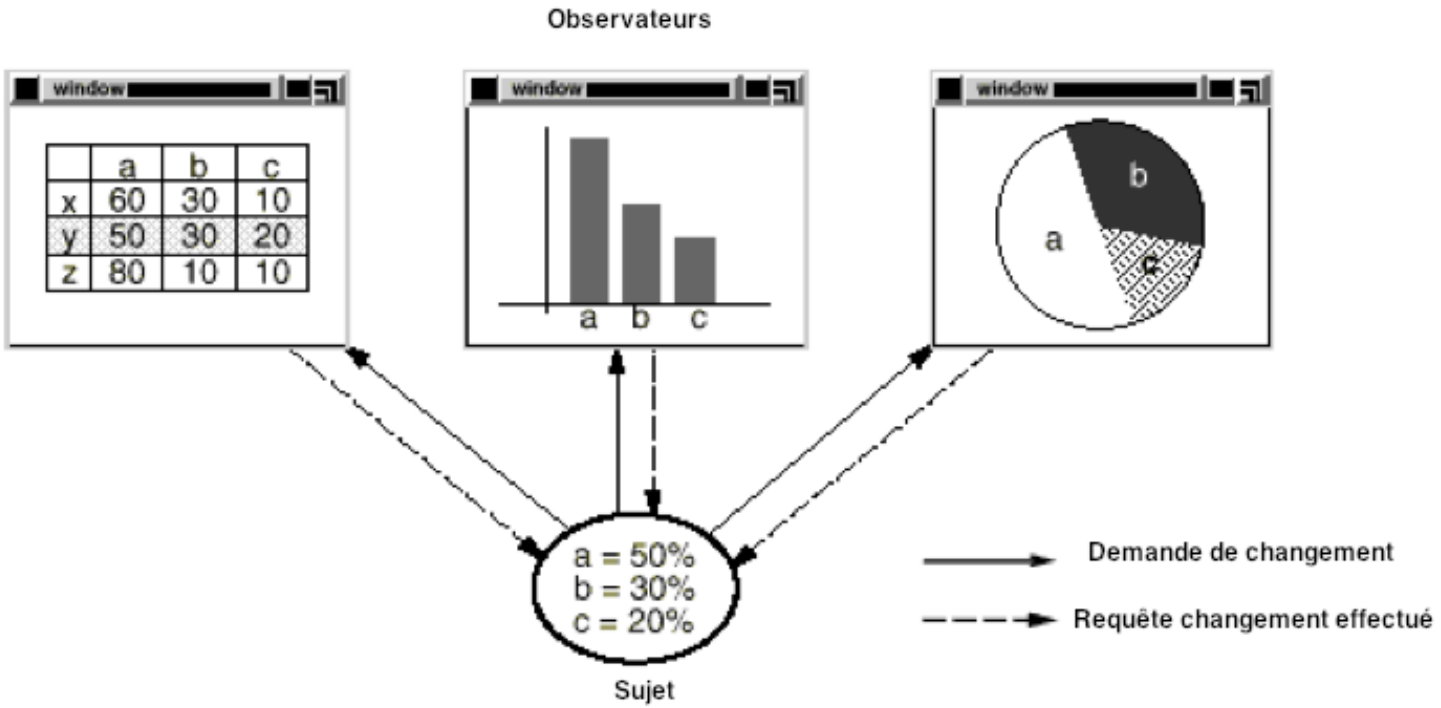
Le Modèle observateur (The Observer Pattern).

Problème :

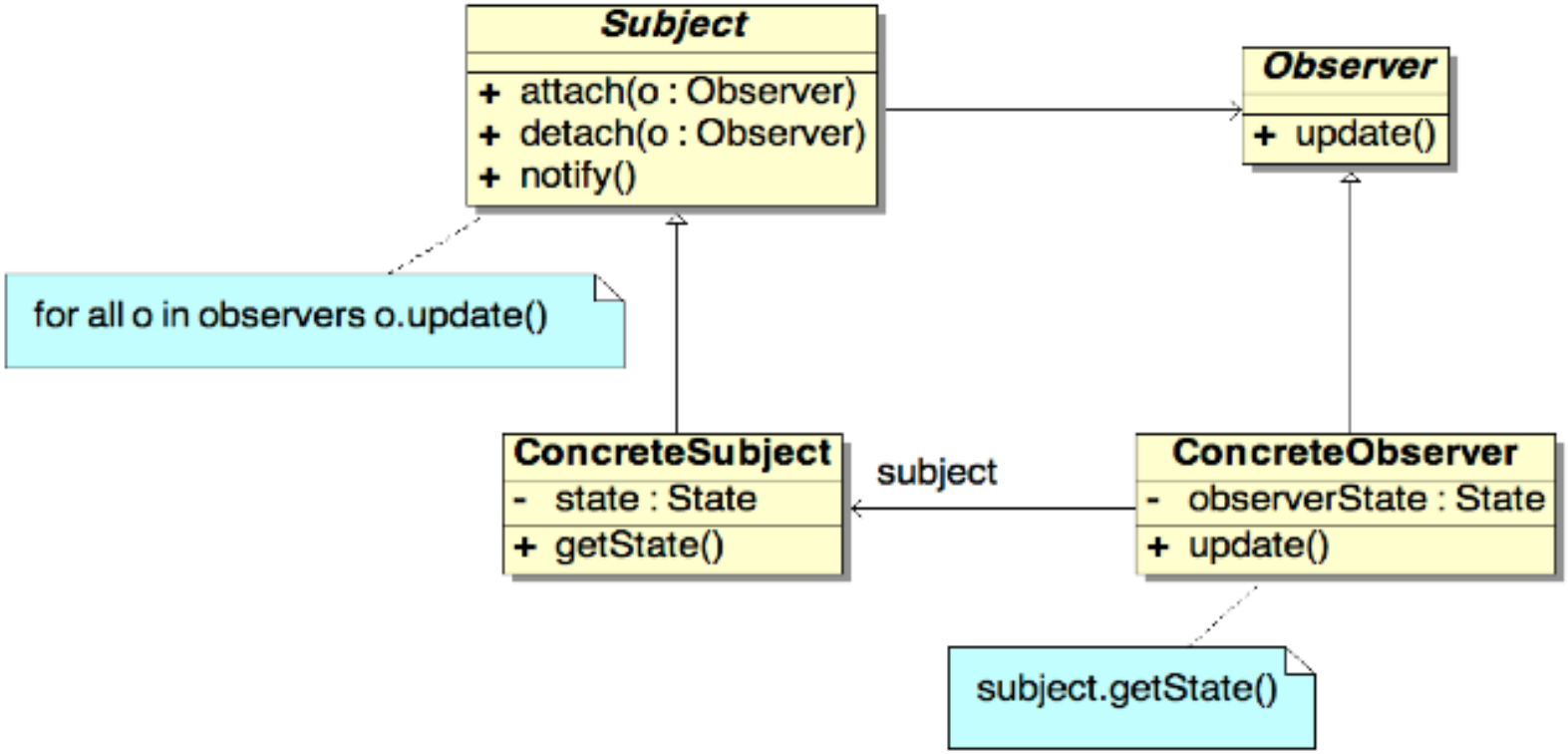
On utilise ce design pattern pour :

- Garder la cohérence entre des classes coopérant entre elles toute en maintenant leur indépendance.
- définir la dépendance one-to-many c-a-d en changeant l'état d'un objet la objets qui lui sont dépendant se mettent automatiquement a jour.
- Ajouter des observateurs nouveaux avec un minimum de changement.

Les Patterns de Comportement : Observer



Les Patterns de Comportement : Observer



Subject – Connais les observateurs.
– Interface pour attacher et détacher les observateurs

Observer – Définis l'interface de mise à jour des objets qui doivent être alerter des changements dans Subject.

ConcreteSubject – Stocke les états qui interesse l'objet de ConcreteObserver.
– Envois une notification aux observateurs en cas de changement d'état.

ConcreteObserver – garde les réferences aux objets de ConcreteSubject.

Les Patterns de Comportement : Observer

Conséquences :

- Avantages : Couplage abstrait entre un sujet et un observateur, support pour la communication par diffusion.
- Limitations : Des mises à jour inattendues peuvent survenir avec des coûts importants.

Exemples :

- Modèle-vue-contrôleur : utiliser pour séparer le modèle (observable) de la vue(observer).
- Manager d'évènement : Swing and .net utilisent le pattern observer pour implémenter le mécanisme d'évènement.

Les Patterns de Comportement : Observer

Exemple : Agence d'information

- L'agence collecte les informations et les publies a ses abonnés.
- Une fois l'information récupérée, le système dois l'envoyer immédiatement aux abonnés.
- Les abonnés peuvent recevoir les informations de différentes manières : Emails, SMS, ...
- Le système doit être facilement extensible afin d'intégrer de nouvelles manières de diffusions d'informations.

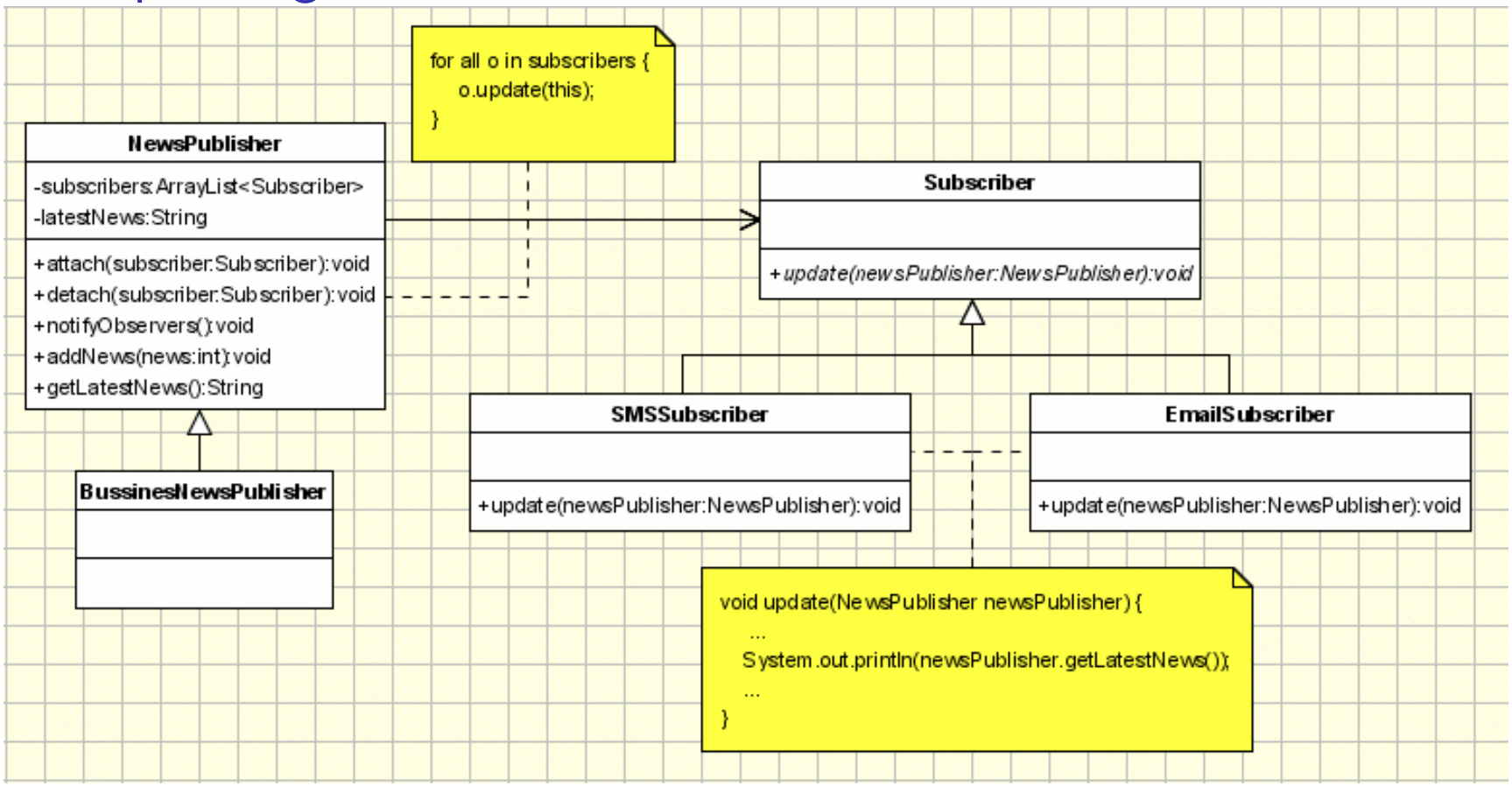
Les Patterns de Comportement : Observer

Exemple : Agence d'information

- L'agence collecte les informations et les publies a ses abonnés.
- Une fois l'information récupérée, le système dois l'envoyer immédiatement aux abonnés.
- Les abonnés peuvent recevoir les informations de différentes manières : Emails, SMS, ...
- Le système doit être facilement extensible afin d'intégrer de nouvelles manières de diffusions d'informations.

Les Patterns de Comportement : Observer

Exemple : Agence d'information



Les Patterns de Comportement : Observer

Exemple : Agence d'information

- La classe **Sujet**(observable) $\Longrightarrow$ **NewsPublisher** : c'est une classe abstraite, sa classe concrète est représentée par **BusinessNewsPublisher**.
- La classe **Observer** $\Longrightarrow$ **Subsciber** : c'est la classe abstraite connue par **NewsPublisher**. Les classe **SMSSubsciber** et **EmailSubsciber** sont les classes concrètes de **Subsciber**.
- Les abonnés peuvent recevoir les informations de différentes manières : Emails, SMS, ...
- Le système doit être facilement extensible afin d'intégrer de nouvelles manières de diffusions d'informations.

Les Patterns de Comportement : Stratégie

Nom :

Le Modèle de stratégie (The Strategy Pattern).

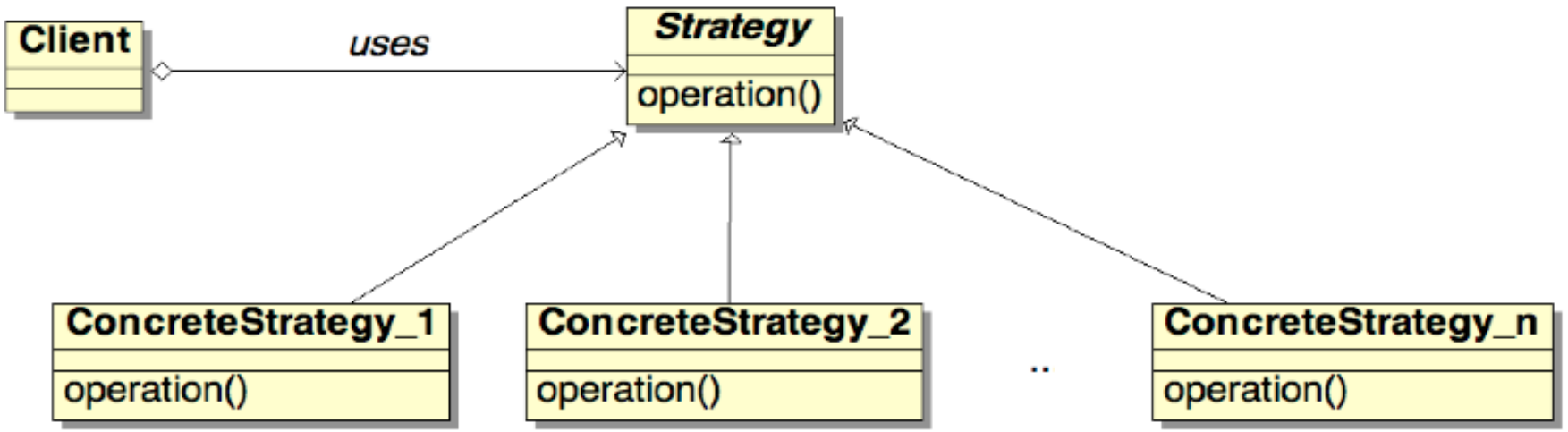
Problème :

On utilise ce design pattern pour :

- Définir une familles d'algorithmes.
- Encapsuler les algorithmes et les rendre interchangeables toute en assurant que chaque algorithme peut évoluer indépendamment des clients qui l'utilisent.

Les Patterns de Comportement : Stratégie

Solution :



Strategy Déclare une interface commune aux differents algorithmes.

ConcreteStrategy –Implémente l'algorithme en utilisant l'interface Strategy.

Client

- est configuré avec un objet de ConcreteStrategy
- Contient une référence vers l'objet Strategy
- Définis une interface qui laisse Strategy accéder aux données.

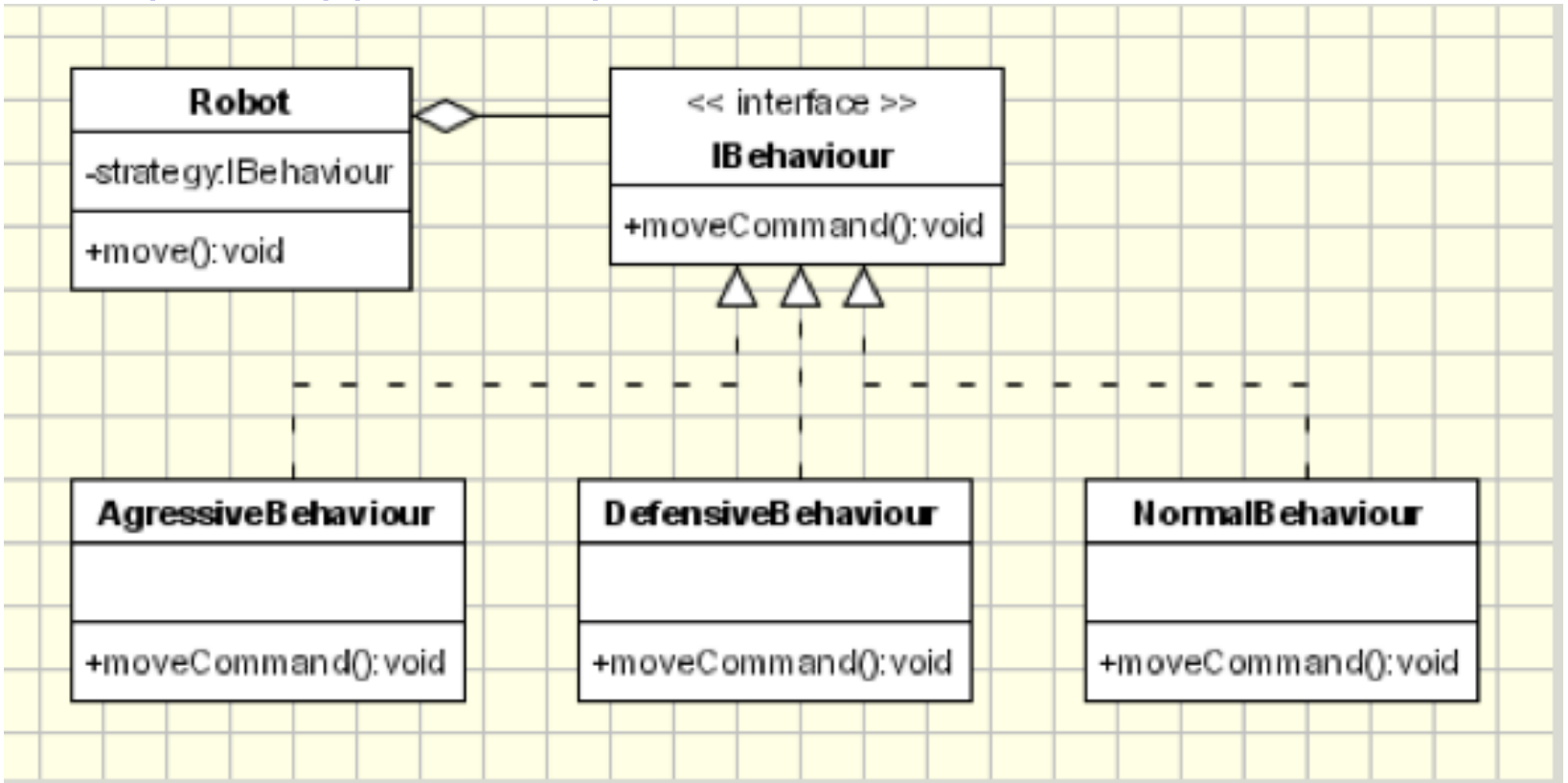
Les Patterns de Comportement : Stratégie

Conséquences :

- Avantages :
  - Expression hiérarchique de familles d'algorithme.
  - Élimination de tests pour sélectionner le bon algorithme.
  - Sélection dynamique de l'algorithme.
- Limitations : Le client doit faire attention à la stratégie, surcout lié à la communication entre stratégies et client.

Les Patterns de Comportement : Stratégie

Exemple : Application pour Robots



Les Patterns de Comportement : Stratégie

Exemple : Application pour Robots

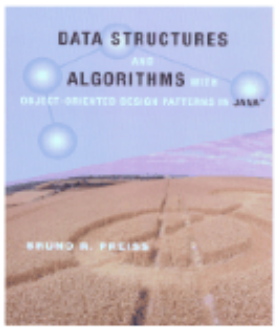
```
public class Main {

        public static void main(String[] args) {

                Robot r1 = new Robot("Big Robot");
                Robot r2 = new Robot("George v.2.1");
                Robot r3 = new Robot("R2");

                r1.setBehaviour(new AgressiveBehaviour());
                r2.setBehaviour(new DefensiveBehaviour());
                r3.setBehaviour(new NormalBehaviour());

                r1.move();
                r2.move();
                r3.move();

                System.out.println("\r\nNew behaviours: " +
                                "\r\n\t'Big Robot' gets really scared" +
                                "\r\n\t, 'George v.2.1' becomes really mad because" +
                                "it's always attacked by other robots" +
                                "\r\n\t and R2 keeps its calm\r\n");

                r1.setBehaviour(new DefensiveBehaviour());
                r2.setBehaviour(new AgressiveBehaviour());

                r1.move();
                r2.move();
                r3.move();
        }
}
```

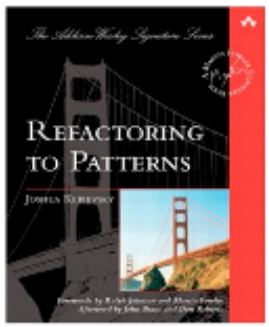
Conclusion

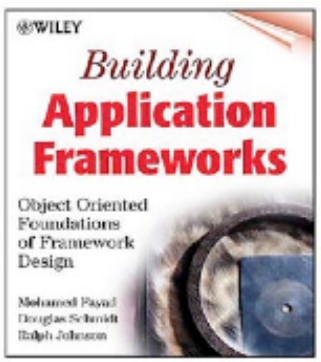
Les designs patterns ..

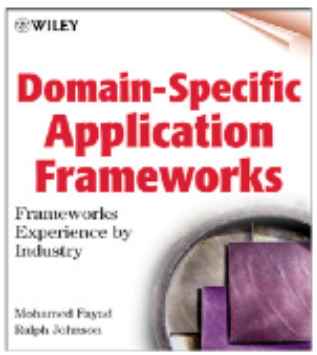
- C'est :
  - Une description d'une solution classique à un problème récurent.
  - une description d'une partie de la solution... avec des relations avec le système et les autres parties...
  - une technique d'architecture logicielle
- Ce n'est pas :
  - Une brique : un pattern dépend de son environnement.
  - Une règle : un pattern ne peut pas s'appliquer mécaniquement.
  - une méthode : ne guide pas une prise de décision : un pattern est la décision prise

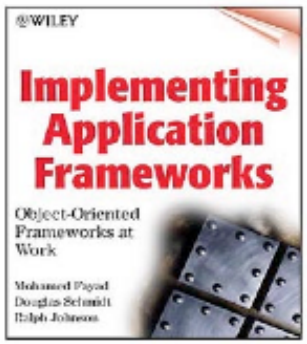
Bibliographie







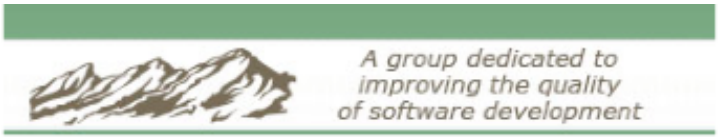




Bibliographie

- " Pattern Languages of Program Design ", Coplien J.O., Schmidt D.C., Addison-Wesley, 1995.
- " Pattern languages of program design 2 ", Vlissides, et al, ISBN 0-201-89527-7, Addison-Wesley
- " Pattern-oriented software architecture, a system of patterns ", Buschmann, et al, Wiley
- " Advanced C++ Programming Styles and Idioms ", Coplien J.O., Addison-Wesley, 1992.
- S.R. Alpert, K.Brown, B.Woolf (1998) The Design Patterns Smalltalk Companion, Addison-Wesley (Software patterns series).
- J.W.Cooper (1998), The Design Patterns Java Companion, http:// www.patterndepot.com/put/8/JavaPatterns.htm.
- S.A. Stelting, O.Maasen (2002) Applied Java Patterns, Sun Microsystems Press.
- Communications of ACM, October 1997, vol. 40 (10).
- Thinking in Patterns with Java http://mindview.net/Books/TIPatterns/

Bibliographie



- http://hillside.net/
- Portland Pattern Repository
  - http://www.c2.com/ppr
- A Learning Guide To Design Patterns
  - http://www.industriallogic.com/papers/learning.html
- Vince Huston
  - http://home.earthlink.net/~huston2/
- Ward Cunnigham's WikiWiki Web
  - http://www.c2.com/cgi/wiki?WelcomeVisitors
- Core J2EE Patterns
  - http://www.corej2eepatterns.com/index.htm