IFT3912 – Développement et maintenance de logiciels

Patrons de conception

Bruno Dufour
Université de Montréal
dufour@iro.umontreal.ca

Patrons de conception (*design patterns*)

- Un patron décrit (et nomme) une solution à un problème commun
  - Offre une solution abstraite pour faciliter la réutilisation
  - Est formulé en termes de classes et d'objets
  - Peut être implémenté différemment en fonction du langage de programmation utilisé
- La solution proposée par un patron peut être modifiée ou adaptée selon les besoins du logiciel
  - « Forcer » l'utilisation d'un patron dans un logiciel est une mauvaise pratique de développement

Objectifs des patrons

- Les patrons visent en général à accroître la qualité du code en visant un ou plusieurs des objectifs suivant:
  - Flexibilité accrue
  - Meilleure performance
  - Fiabilité accrue
- Attention! L'utilisation des patrons peut aussi augmenter la complexité du code
  - Par exemple, ajout d'indirections
  - Il faut donc juger des avantages et inconvénients de l'ajout de patrons dans la conception d'un logiciel

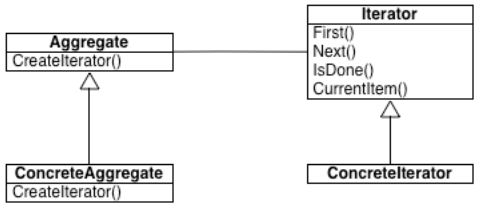
Types de patrons

- Créationnels: Abstract Factory, Singleton, Builder, …
  - Font l'abstraction du processus d'instanciation afin de rendre un système indépendant de la façon dont ses objets sont créés et représentés
- Structurels: Adapter, Bridge, Composite, Decorator, Façade, …
  - Se concentrent sur la façon dont les classes et les objets sont composés pour obtenir de plus grandes structures
- Comportementaux: Iterator, Visitor, Observer, State, Strategy, Mediator, …
  - Décrivent les modèles de communication et interaction entre les objets

4

Iterator (Comportemental)

- Problème
  - Fournir un accès séquentiel aux éléments d'un groupe sans exposer la représentation interne
  - Supporter plusieurs types d'accès
  - Supporter les accès concurrents
- Solution:



5

Iterator - Exemple

```
public class ArrayList {
    private Object[] data;
    private int size;

    public Iterator iterator() {
        return new Iterator() {
            private int index;

            public void first() {
                index = 0;
            }

            public void next() {
                index++;
            }

            public boolean done() {
                return index >= size;
            }

            public Object current() {
                if (done()) {
                    throw new NSEException();
                }
                return data[index];
            }};
    }
}
```

```
public interface Iterator {
    public void first();
    public void next();
    public boolean done();
    public Object current();
}
```

6

Iterator – Exemple simplifié

```
public class ArrayList {
    private Object[] data;
    private int size;

    public Iterator iterator() {
        return new Iterator() {
            private int index = 0;

            public Object next() {
                if (!hasNext()) {
                    throw new NSEException();
                }
                return data[index++];
            }

            public boolean hasNext() {
                return index < size;
            }
        };
    }
}
```

```
public interface Iterator {
    public Object next();
    public boolean hasNext();
}
```

7

Iterator - Implémentation

- Qui contrôle l'itération?
  - Le client: itérateur externe
  - L'itérateur: itérateur interne
- Qui définit l'algorithme de traverse?
  - L'objet Aggregate
  - L'itérateur (peut violer les principes d'encapsulation)
- L'itérateur est-il robuste en présence de modifications du groupe?
  - Sans copier le groupe!
- Accès privilégié
- Itérateur « nul »

8

Singleton (Créationnel)

- Problème
  - Il est souvent important pour une classe de n'avoir qu'une instance (facilement accessible).
  - Une variable globale n'est pas suffisamment flexible
- Solution:
  - Assurer une instance unique en cachant le mécanisme de création (constructeur privé en Java)
  - Garder une référence pour l'instance unique (attribut statique privé)
  - Créer un point d'accès publique (une méthode qui retourne l'instance unique)

9

Singleton (Créationnel) - Exemple

```
public class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton () {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

10

Les patrons peuvent être adaptés

```
public class LazySingleton {
    private static LazySingleton instance;

    private LazySingleton() {}

    public static LazySingleton getInstance() {
       if (instance == null) {
             instance = new LazySingleton();
       }
         return instance;
    }
}
```

11

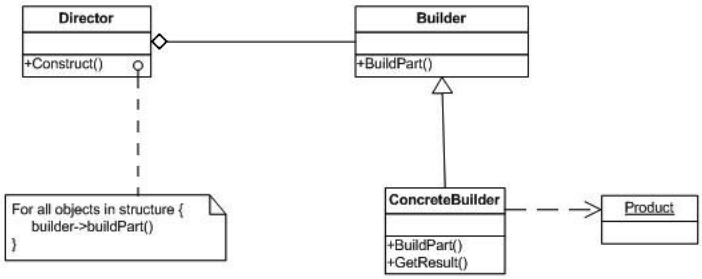
Les patrons peuvent être adaptés (2)

```
public class ThreadSingleton {
    private static ThreadLocal instances
            = new ThreadLocal();

    private ThreadSingleton() {}

    public static ThreadSingleton getInstance() {
        ThreadSingleton instance = (ThreadSingleton) instances.get();
       if (instance == null) {
            instance = new ThreadSingleton();
            instances.set(instance);
       }

       return instance;
    }
}
```

12

Builder (Créationnel)

- Problème
  - Séparer la construction d'un objet complexe de sa représentation de façon à pouvoir construire différentes représentations à partir d'un même processus de création.
- Solution



13

Builder – Exemple (Performance)

```
public String greet(String speaker, String action, Object
consequence) {
    StringBuilder builder = new StringBuilder();

    builder.append("Hello. My name is ");
    builder.append(speaker);
    builder.append(". ");

    builder.append("You ");
    builder.append(action);
    builder.append(". ");

    builder.append("Prepare to ");
    builder.append(consequence);
    builder.append(".");

    return builder.toString();
}

greet("Inigo Montoya", "killed my father", Consequences.DIE);
```

Builder – Exemple (Facilité d'utilisation)

```
public class JavaCommandBuilder {
     private ArrayList<String> args;

     public JavaCommandBuilder(String java) {
          this.args = new ArrayList<String>(5);
          this.args.add(java);
     }

     public JavaCommandBuilder setMaxHeap(String heapSize) {
          this.ensureStage(PRE_MAIN, "max heap");
          this.args.add("-X" + heapSize);
          return this;
     }

     public JavaCommandBuilder setClasspathToCurrent() {
          return this.setClasspath(System.getProperty("java.class.path"));
     }

     public JavaCommandBuilder setClasspath(String classpath) {
          this.ensureStage(PRE_MAIN, "classpath");
          this.args.add("-classpath");
          this.args.add(classpath);
          return this;
     }

     public JavaCommandBuilder setClasspath(String[] classpath) {
          return this.setClasspath(Strings.join(classpath, File.pathSeparator));
     }
```

Builder – Exemple (Facilité d'utilisation)

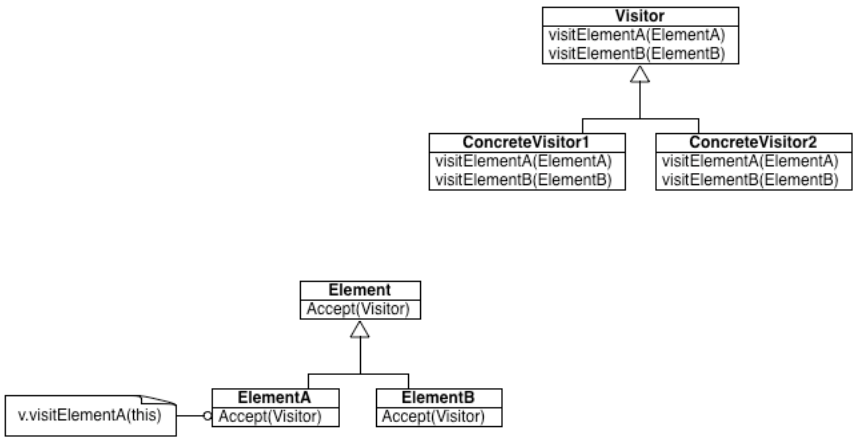
```
     public JavaCommandBuilder setMainClass(String className) {
          this.ensureStage(PRE_MAIN, "main class");
          this.args.add(className);
          this.stage = POST_MAIN;
          return this;
     }

     public JavaCommandBuilder setMainClass(Class<?> c) {
          return this.setMainClass(c.getName());
     }

     public JavaCommandBuilder setMainJar(File jar) {
          this.ensureStage(PRE_MAIN, "main jar");
          this.args.add("-jar");
          this.args.add(jar.getAbsolutePath());
          this.stage = POST_MAIN;
          return this;
     }

     public JavaCommandBuilder addArgument(String argument) {
          this.ensureStage(POST_MAIN, "argument");
          this.args.add(argument);
          return this;
     }

     public String[] build() {
          if (this.stage != POST_MAIN) {
               throw new IllegalStateException("No main class or jar added");
          }

          return this.args.toArray(new String[this.args.size()]);
     }
```

Visitor (Comportemental)

- Problème
  - Effectuer une opération sur les éléments d'une structure sans modifier la classe des éléments
- Solution
  - Définir une interface Visitor
  - Ajouter une méthode visitX à l'interface pour chaque type d'élément
  - Ajouter une méthode accept(Visitor) à chaque élément (en général, à une class parente de tous les éléments
  - Pour chaque type d'élément, implémenter accept(Visitor) pour appeler Visitor.visitX(this)

17

Visitor – Diagramme de classes



18

Visitor - Exemple

```
public interface FSVisitor {
    public void visitFile(File f);
    public void visitDirectory(Directory d);
}
```

19

Visitor - Exemple

```
public class File {
    @Override
    public void accept(FSVisitor v) {
        v.visitFile(this);
    }
}

public class Directory {
    public File[] getFiles() {...}
    public Directory[] getSubDirectories() {...}

    @Override
    public void accept(FSVisitor v) {
        v.visitDirectory(this);
    }
}
```

20

Visitor – Exemple

```
public class FSCountingVisitor implements FSVisitor {
    private int files = 0;
    private int directories = 0;

    public void visitFile(File f) {
        files += 1;
    }

    public void visitDirectory(Directory dir) {
        directories += 1;
       for (File f: dir.getFiles()) f.accept(this);
       for (Directory d: dir.getSubDirectories()) {
           d.accept(this);
       }
    }

    public void count(Directory root) {
        root.accept(this);
    }
}
```

21

Visitor – Exemple (alternative)

```
public class File {
    @Override
    public void accept(FSVisitor v) {
        v.visitFile(this);
    }
}

public class Directory {
    private File[] files;
    private Directory[] subdirs;

    @Override
    public void accept(FSVisitor v) {
        v.visitDirectory(this);
       for (File f: files) f.accept(v);
       for (Directory d: subdirs) d.accept(v);
    }
}
```

22

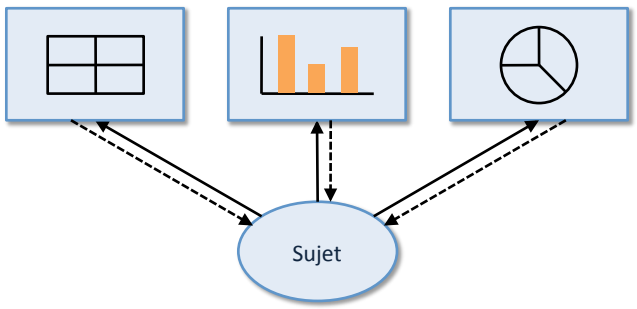
Visitor – Exemple (alternative)

```
public class FSCountingVisitor implements FSVisitor {
    private int files = 0;
    private int directories = 0;

    public void visitFile(File f) {
        files += 1;
    }

    public void visitDirectory(Directory d) {
        directories += 1;
    }

    public int getFileCount() {
        return files;
    }

    public int getDirectoryCount() {
        return directories;
    }

    public void count(Directory root) {
        root.accept(this);
    }
}
```

23

Observer (Comportemental)

- Problème
  - Créer une dépendance un-à-plusieurs entre des objets de sorte que lorsque un objet change, tous ses dépendants sont notifiés et mis à jour



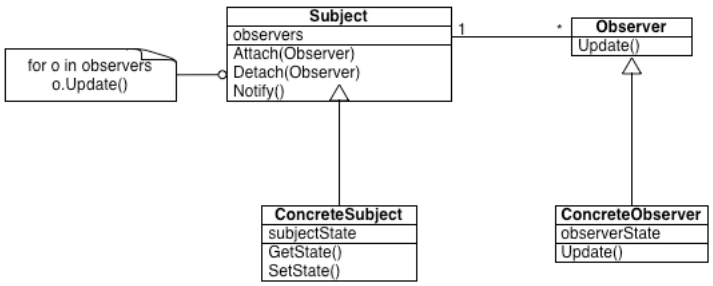
24

Observer (Comportemental)

- Solution
  - Définir une interface *Sujet* pour l'objet principal
  - Définir une interface pour les objets *observateurs*
    - Définir une méthode *update* pour les observateurs qui sera appelée lors de changements du sujet
  - Définir un protocole pour que les observateurs puissent s'enregistrer auprès du sujet
  - S'assurer que le sujet appelle *update()* lors de changements

25

Observer – Diagramme de classes



26

Observer - Exemple

```
public interface Observer {
    public void update();
}

public abstract class Subject {
    private List<Observer> observers;

    public void attach(Observer o) {
        observers.add(o);
    }

    public void detach(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers() {
        for (Observer o: observers) {
            o.update();
        }
    }
}
```

27

Observer - Exemple

```
public class Timer extends Subject {
    public int getHours() { ... }
    public int getMinutes() { ... }
    public int getSeconds() { ... }

    public void tick() {
        // update internal state
        notifyObservers();
    }
}
```

28

Observer - Exemple

```
public class DigitalClock implements Observer {
    private Timer timer;

    public DigitalClock(Timer timer) {
        this.timer = timer;
        timer.attach(this);
    }

    protected void finalize() throws Throwable {
        timer.detach(this);
    }

    public void update() {
        int hours = timer.getHours();
        int minutes = timer.getMinutes();
        int seconds = timer.getSeconds();

        // update display
    }
}

public class AnalogClock implements Observer {...}
```

29

Factory - Motivation

```
public class PizzaStore {
    public Pizza orderPizza() {

        Pizza pizza = new Pizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }
}
```

Source: Freeman et. al., "Head First Design Patterns", O'Reilly 2004.

Factory - Motivation

```
public class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("greek")) {
            pizza = new GreekPizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        }

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }
}
```

Source: Freeman et. al., "Head First Design Patterns", O'Reilly 2004.

Factory - Motivation

```
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("greek")) {
            pizza = new GreekPizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        }

        return pizza;
    }
}
```

Source: Freeman et. al., "Head First Design Patterns", O'Reilly 2004.

Factory - Motivation

```
public class PizzaStore {
    private SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }
}
```

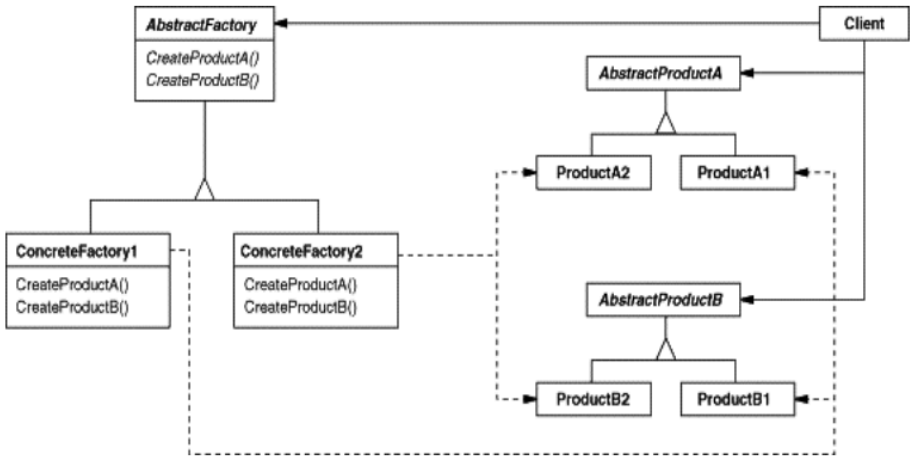
Source: Freeman et. al., "Head First Design Patterns", O'Reilly 2004.

Abstract Factory (Créationnel)

- Problème
  - Il est souvent nécessaire de créer de familles d'objets reliés ou dépendants sans spécifier leur classe concrètement
    - Ex: changer l'apparence des éléments d'interface graphique

34

Abstract Factory - Solution



Abstract Factory - Exemple

```
public interface WidgetFactory {
    public Button newButton(String label);
    public Label newLabel(String text);
    public TextField newTextField();
}

public class GTKWidgetFactory implements WidgetFactory { ... }
public class MacOSWidgetFactory implements WidgetFactory { ... }
public class MSWinWidgetFactory implements WidgetFactory { ... }

public class ApplicationWindow {
    public void create(WidgetFactory factory) {
        Button okButton = factory.newButton("OK");
        Button cancelButton = factory.newButton("Cancel");
        TextField text = factory.newTextField();
        ...
    }
}
```

Factory Method - Motivation

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("cheese");

ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("greek");
```

Source: Freeman et. al., "Head First Design Patterns", O'Reilly 2004.

Factory - Motivation

```
public class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    protected abstract Pizza createPizza(String type);
}
```

Source: Freeman et. al., "Head First Design Patterns", O'Reilly 2004.

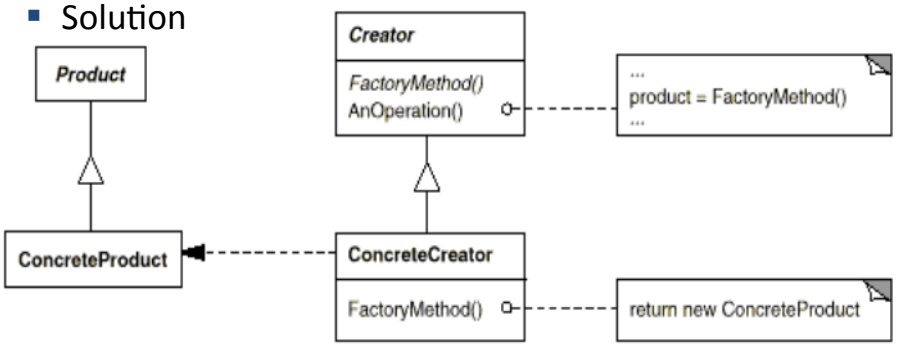
Factory - Motivation

```
public class NYPizzaStore extends PizzaStore {
    protected Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            pizza = new NYCheesePizza();
        } else if (type.equals("greek")) {
            pizza = new NYGreekPizza();
        } else if (type.equals("pepperoni")) {
            pizza = new NYPepperoniPizza();
        }
    }
}

public class NYCheesePizza extends Pizza {…}
…

public class ChicagoPizzaStore extends PizzaStore {…}
```

Source: Freeman et. al., "Head First Design Patterns", O'Reilly 2004.

Factory method (créationnel)

- Problème
  - Une classe est incapable d'anticiper le type d'objets qu'elle doit créer
  - Une classe désire laisser le choix du type d'objets créés à ses sous-classes
- Solution

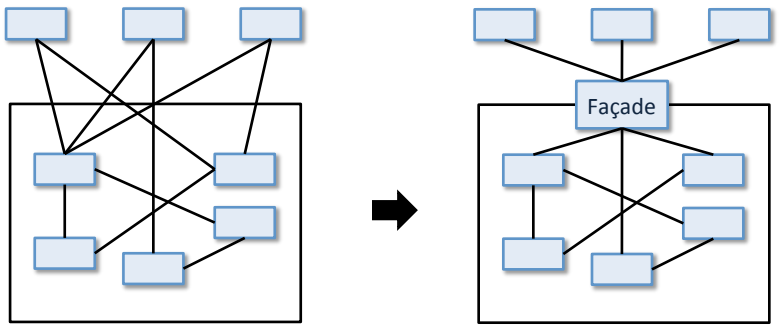


Factory Method - Exemple

```
public interface Document {...}

public abstract class Application {
    public Document newDocument() { ... }
    public Document openDocument() { ... }
    protected abstract Document createDocument();
}

public class MyApplication extends Application {
    protected Document createDocument() {
        return new PlainTextDocument();
    }
}
```

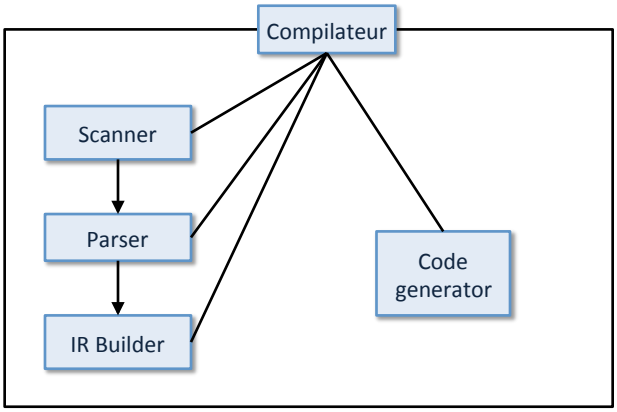
Façade (Structurel)

- Problème
  - Fournir une interface unie pour l'ensemble des interfaces d'un sous-système afin de réduire la complexité tout en maintenant la fonctionnalité
- Solution



42

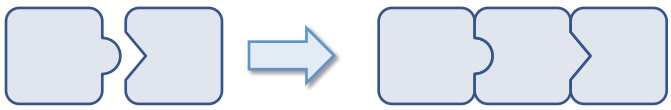
Façade - Exemple



43

Adapter (Structurel)

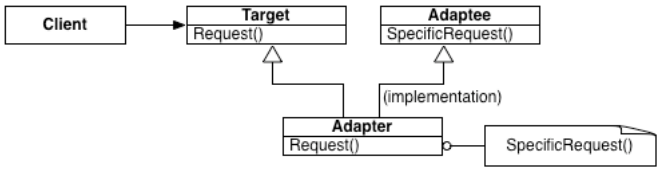
- Problème
  - Convertir l'interface d'une classe en une autre interface attendue par le client (interface cible) afin de permettre à des classes incompatibles de travailler de concert
    - Souvent motivé par la réutilisation de code: le code réutilisé doit se conformer à une interface requise



- Solutions
  - Par classe: héritage multiple / interfaces
  - Par objet: composition

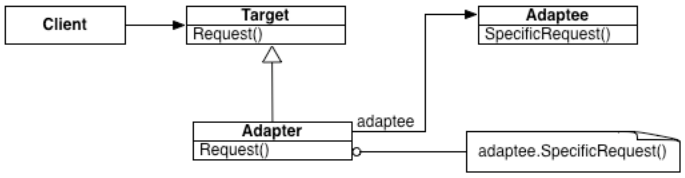
44

Adapter – Par classe



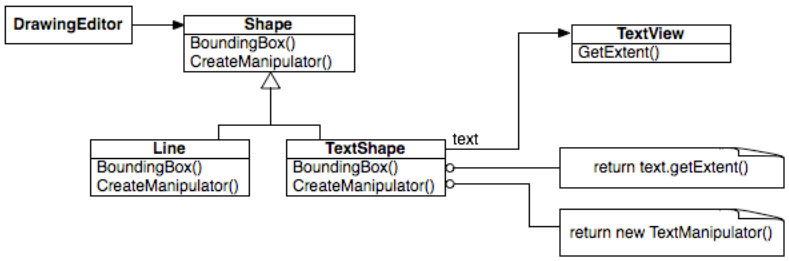
45

Adapter – Par objet



46

Adapter - Exemple



47

Adapter - Exemple

```
public interface Enumeration<T> {
    public boolean hasMoreElements();
    public T nextElement();
}

public interface Iterator<T> {
    public boolean hasNext();
    public T next();
    public void remove();
}
```

Adapter - Exemple

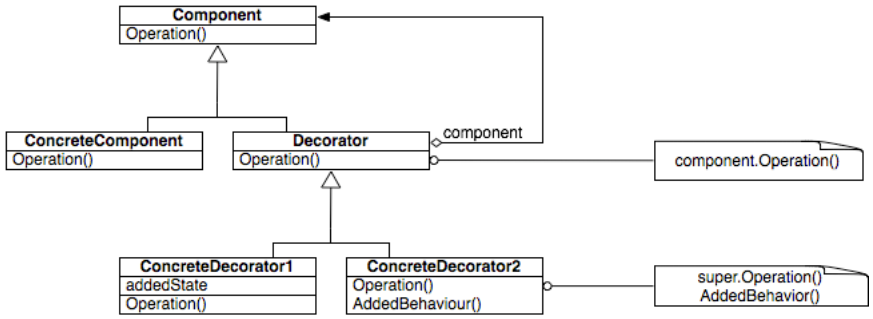
```
public class EnumerationIterator<T> implements Iterator<T> {
    private Enumeration<T> e;

    public EnumerationIterator(Enumeration<T> e) {
        this.e = e;
    }

    public boolean hasNext() {
        return e.hasMoreElements();
    }

    public T next() {
        return e.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Decorator (Structurel)

- Problème
  - Ajouter des responsabilité à un objet durant l'exécution
    - Alternative à l'héritage
- Solution
  - Définir un objet *décorateur*
    - Se conforme à l'interface de l'objet décoré pour transparence
    - Réachemine les requêtes vers l'objet décoré
    - Peut ajouter des fonctions
    - Peut être utilisé une chaîne de décorateurs

50

Decorator – Diagramme de classes



51

Decorator - Exemple

52

Decorator - Implémentation

- Omission de la classe abstraite Decorator
- Garder la classe Component aussi simple que possible
- Changer l'extérieur d'un objet vs l'intérieur
  - Extérieur = Decorator
  - Intérieur = Strategy

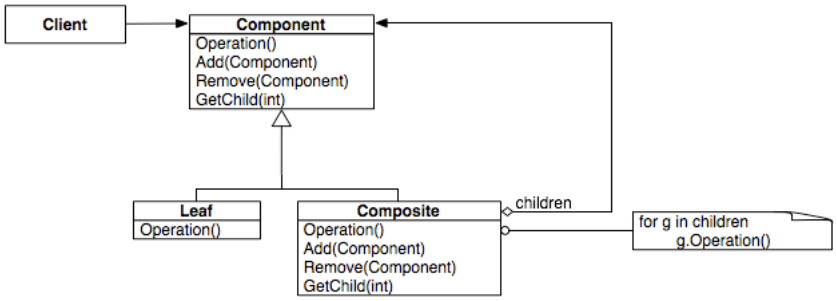
53

Composite (Structurel)

- Problème
  - Composer des objets en une hiérarchie de sorte que les objets individuels et composés sont traités de manière uniforme
    - Composition récursive
- Solution
  - Définir une class abstraite (ou interface) qui représente à la fois les objets simples individuels et les objets composés

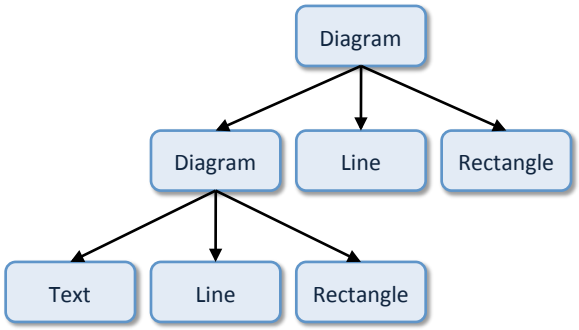
54

Composite – Diagramme de classes



55

Composite - Exemple



56