IFT2255 - Génie logiciel

Conception

Bruno Dufour
dufour@iro.umontreal.ca

L'activité de conception

L'activité de conception

- Étape cruciale du développement logiciel
  - analyse → conception → implémentation
  - décomposition du produit en unités (ex: modules, classes) plus simples
- Difficile: exige de la créativité
- Un bonne conception contribue à la qualité du logiciel
  - fiabilité
  - correction
  - facilité d'évolution et maintenance

Types de conception

- Conception architecturale (haut niveau)
  - Définit la structure et l'organisation générale du logiciel demandé
  - Décrit les principaux modules, les relations entre eux, les contraintes à respecter
- Conception détaillée (bas niveau)
  - Réalise chacun des cas d'utilisation
  - Respecte le plan de la conception architecturale
  - Décrit le fonctionnement interne de chacun des modules
  - Permet l'implémentation dans un language de programmation

Objectifs d'une bonne conception

- **Forte cohésion** : les éléments ne sont pas réunis dans un même module (ou une même classe) par hasard, ils forment un tout pour réaliser une tâche
- **Faible couplage** : les modules (ou classes) sont relativement indépendants, ils ne dépendent le moins possible des éléments d'autres modules
- **Abstraction** : une décomposition intuitive et qui permet de se concentrer sur un module (ou classe) à la fois
- **Encapsulation et masquage d'information** : les détails d'implémentation sujets au changements sont cachés derrière une interface stable.

Conception et maintenance

- La conception doit tenir de compte
  - des besoins existants
  - des besoins à venir
- Objectif : obtenir une conception qui facilite l'adaptation du logiciel aux changements
  - capacité d'évolution
  - anticipation du changement

Types de changements

- Changement d'algorithme / de fonctionnalité
- Changement de la représentation des données
- Changement au niveau de la machine abstraite sous-jacente
- Changement au niveau des périphériques
- Changement de l'environnement
- Changements dus au processus de développement
- ...

Modularisation

Modules

- **Objectif** : déterminer la structure modulaire du logiciel à développer
- **Module** : unité fournissant des ressources et/ou des services
  - Quels sont les modules ?
    - $S = \{ M_1, M_2, ..., M_n \}$
  - Quelles relations lient les modules entre eux ?
    - $r \subseteq S \times S$

Relations entre modules

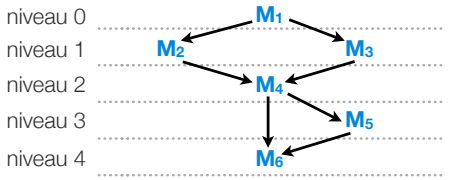
- Deux types de relations sont utiles pour décrire la structure d'un système
  - **Relation « contient »** : regroupement en paquetages, classes internes (aussi agrégation/ composition)
  - **Relation « utilise »** : association, agrégation/ composition, généralisation, liens de dépendance

Relation « utilise »

- M_i utilise M_j si M_i requiert la présence de M_j car M_j lui fournit des ressources ou des services
- Idéalement une hiérarchie :
  - Facilite la compréhension de la structure (par niveau d'abstraction)
  - Facilite le test unitaire
  - Permet de bâtir un système partiel mais fonctionnel

Relation « utilise »

- Définit des niveaux d'abstraction



- On définit le niveau d'un module M_i dans une hiérarchie r
  - niveau(M_i) = 0 s'il n'existe pas de M_k tel que $M_k \, r \, M_i$,
  - niveau(M_i) = max({ niveau(M_k) | $M_k \, r \, M_i$ }) + 1 sinon

Interface & implémentation

- **Interface** : partie publique
  - l'ensemble des ressources (opérations, attributs, etc.) rendues accessibles aux modules clients
  - la conception d'un module M ne nécessite que les interfaces des autres modules que M pourra utiliser
- **Implémentation** : partie privée
  - façon dont les ressources sont concrètement représentées et réalisées dans le module
- Séparation des préoccupations
  - Quoi offrir ? → Analyse et conception
  - Comment le réaliser ? → Conception et implémentation

Gestion des anomalies

- État anormal : état d'un module qui ne peut offrir le service tel qu'attendu et spécifié par son interface
- Causes possibles de la défaillance d'un module
  - Service invoqué avec des paramètres inadéquats
  - Préconditions non-satisfaites
  - Utilisation un service défaillant exporté par un autre module
  - etc.

Gestion des anomalies

- En cas d'anomalie, un module offrant un service qui doit :
  - gérer l'anomalie localement **seulement** si approprié
  - signaler l'anomalie en levant une exception auprès du module client sinon
    - Le client se chargera de gérer correctement l'exception, ou répétera le processus
- Les exception qu'un module serveur peut lever doivent être indiquées dans son interface

Architecture logicielle

Tâches de la phase de conception

- Concevoir l'architecture
  - « Qui » fera « quoi » et « où » ?
- Concevoir les interfaces utilisateurs du logiciel
- Concevoir les bases de données
- Concevoir les contrôles du logiciel
  - Mise en œuvre de tous les aspects liés au contrôle, à la correction, à la sécurité, à la tolérance aux fautes, à la protection des données, etc.
- Concevoir le réseau
  - Communication entre les processus distribués

Architecture logicielle

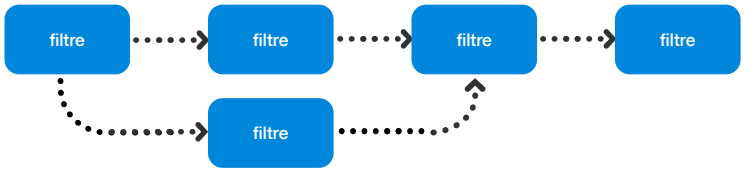
- Décrit
  - L'organisation générale du logiciel
  - Les modules et leurs interfaces
  - L'agencement du logiciel des sous-systèmes, des modules et des composants
    - Leurs propriétés
    - Leur composition («contient »)
    - Leurs collaborations («utilise»)
- Dépend des besoins fonctionnels et non fonctionnels du logiciel

Utilité des architectures

- « Modèles connus » de décomposition éprouvés et enrichis par l'expérience de plusieurs développeurs
- Mode d'interaction établi entre modules via une interface générique
- Fournit une plate-forme d'intégration pour connecter les différentes sous-parties du logiciel à développer
- Meilleure qualité du logiciel : compréhensibilité, maintenance, facilité d'évolution, réutilisation, performance, documentation, etc.

Types d'architectures

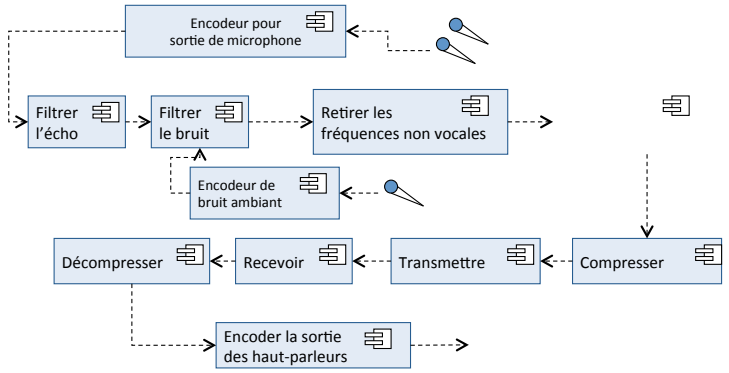
Architecture en pipeline



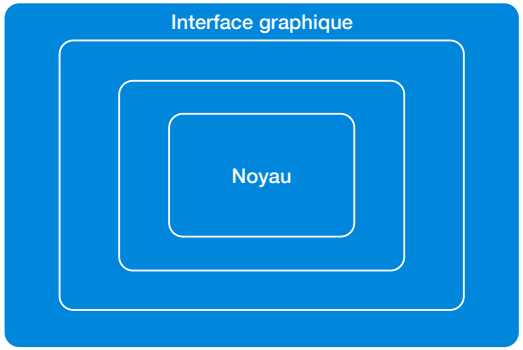
Architecture en pipeline

- Sous-systèmes organisés en pipeline de filtres indépendants
  - La sortie d'un filtre correspond à l'entrée d'un autre
- Communication locale (voisins de gauche et de droite)
  - un filtre peut souvent commencer à opérer avant même d'avoir lu tout le flux d'entrée
- Utile pour les traitements en plusieurs étapes
- Exécution concurrente de filtres possible, synchronisation des flux parfois nécessaire nécessaire

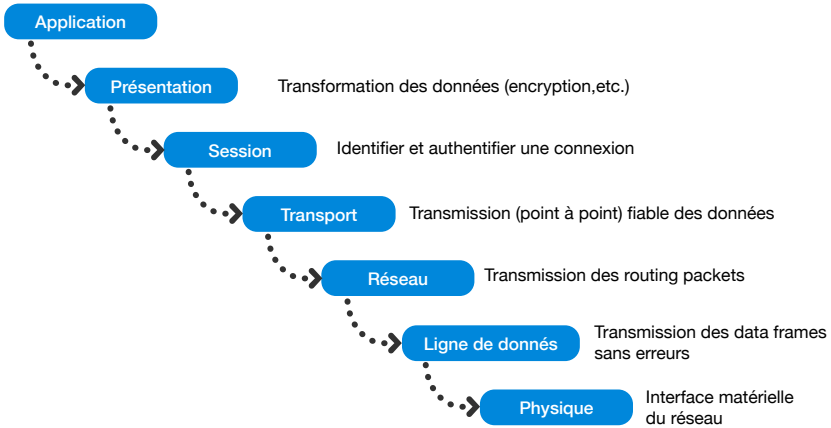
Architecture en pipeline - exemple



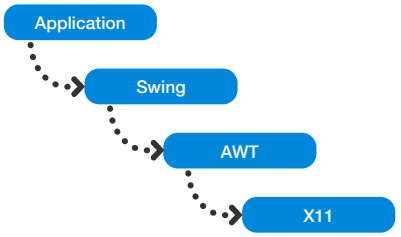
Architecture par couches



Architecture par couches - exemple

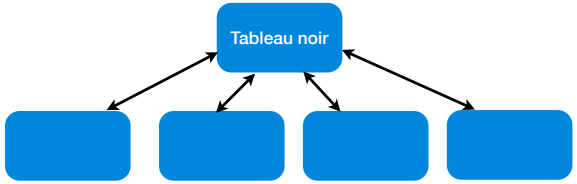


Architecture par couches - exemple



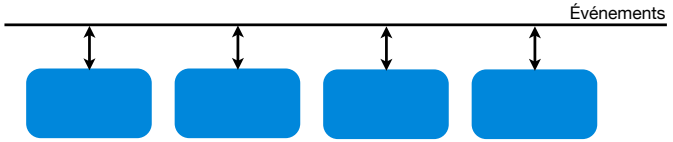
Architecture tableau noir

- Tableau noir : médium de communication
- Communication étendue à tous les partenaires
- Un des sous-système est désigné comme le tableau noir
- On ne peut recevoir et transmettre des informations que via le tableau
- Utile lorsqu'on ne connaît pas de solution déterministe



Architecture par événements

- Les sous-systèmes répondent aux événements
- Appropriée pour les logiciels dont les composants doivent interagir avec l'environnement
- Les sous-systèmes s'abonnent pour recevoir les annonces de certains types d'événements



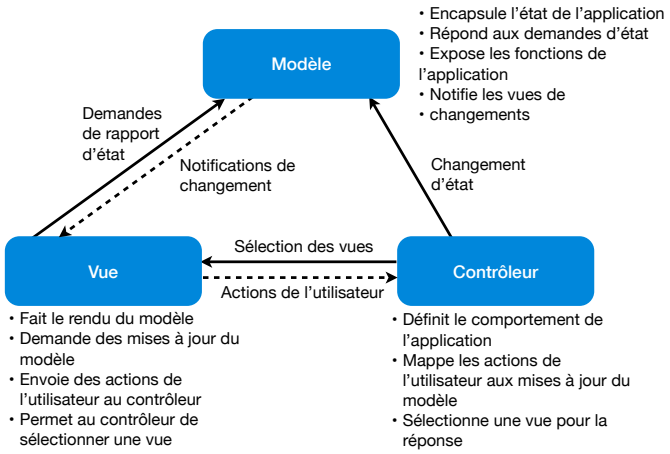
Modèle-vue-contrôleur (MVC)

- Inventé en 1978 par Trygve Reenskaug (U. Oslo)
  - Lors d'une visite à Xerox PARC
- Problème: permettre à des utilisateurs de contrôler de grands et complexes ensembles de données
  - Maximum de flexibilité et de réutilisation possible
- Peut être vu comme
  - Un patron de conception
  - Un type d'architecture
  - Un cadre d'applications (*framework*)

Modèle-vue-contrôleur (MVC)

- Modèle
  - Représente les connaissances dans le système
- Vue
  - Une représentation visuelle du modèle
  - Souligne certains aspects du modèle et en ignore d'autres
- Contrôleur
  - L'interface entre l'utilisateur et le système
  - Responsable des entrées (buttons, etc.) et des sorties (placer les vues sur un moniteur, etc.)

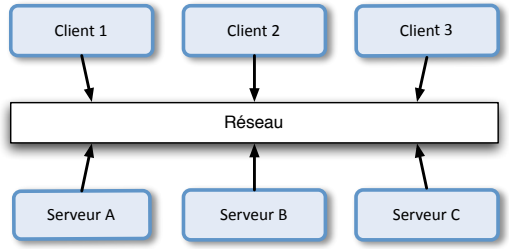
Modèle-vue-contrôleur (MVC)



Architecture client-serveur

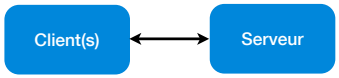
- Clients
  - Reçoivent des services des serveurs
  - Doivent connaître comment contacter les serveurs
  - Ex: adresse IP, port, etc.
- Serveurs
  - Fournissent des services aux clients et autres serveurs
- Extensibilité par ajout de serveurs

Client-serveur



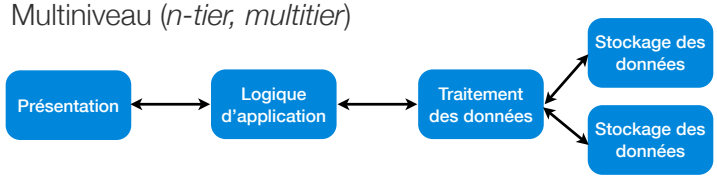
Architecture *n*-parties

- Architecture par niveaux
- 2 niveaux (*2-tier*)



- 3 niveaux (*3-tier*)

- Multiniveau (*n-tier, multitier*)



Types de clients

- Clients légers
  - Le client est responsable de la présentation uniquement
  - Facile à déployer et à maintenir à jour
  - Utilise beaucoup de ressources du(des) serveur(s)
  - Ex: fureteur web (sans javascript, AJAX, etc)
- Clients lourds
  - Le client est responsable de la présentation et de la logique d'application
  - Utilise les ressources du client de manière plus efficace
  - Complexité accrue
  - Ex: Guichet bancaire

Variante - peer-to-peer

- Chaque composant joue le rôle à la fois du client et du serveur
  - Aucun serveur central
- Modèle hybride : un serveur central conserve l'information au sujet des pairs
  - ex: Skype
- Avantages
  - fiabilité (tolérance aux pannes)
  - mise à l'échelle facile (grandit avec le nombre de pairs)
- Inconvénients
  - complexité, rôle plus lourd des pairs

Variante - Cloud computing

- **Cloud computing** : le serveur est géré par un fournisseur de service, et typiquement accessible par internet
  - ex: Amazon Web Services (AWS)
- Avantages
  - Facilité (sécurité, fiabilité, etc.)
  - Mise à l'échelle
- Inconvénient
  - Contrôle réduit

Architecture par objets distribués

- Élimine la distinction entre client et serveur
  - Des objets fournissent l'interface pour les services qu'ils fournissent
  - D'autres objets appellent ces objets sans distinction logique entre objets clients et serveurs
- Les intergiciels assurent la partage des objets entre les différentes machines
  - « Bus » logiciel
  - Gestionnaire ORB (Object Request Broker)

Architecture par objets distribués

- Avantages
  - Possibilité de retarder certaines décisions:
    - Ex: client lourd / léger
  - Architecture ouverte à l'ajout de nouvelles ressources
  - Flexibilité et extensibilité
  - Possibilité de reconfiguration dynamique
    - Ex: des objets serveurs se déplacent vers la même machine que des objets clients qui les utilisent
    - Ex: conversion de 2 niveaux à 3 niveaux après déploiement
- Différents standards pour les intergiciels d'objets distribués:
  - CORBA (Sun, IBM, HP, etc.), DCOM (Microsoft), Java RMI, …

Objets distribués - exemple

- Enterprise Java Beans (EJBs)
  - Permettent le développement rapide d'applications transactionnelles décentralisées
  - 3 types d'objets distribués :
    - Session Bean: représente une session avec un client, avec ou sans état
    - Message Bean: répond à des messages asynchrones
    - Entity Bean: représente des données persistantes (ex: en provenance d'une base de données)

Patrons de conception

Patrons de conception

- 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

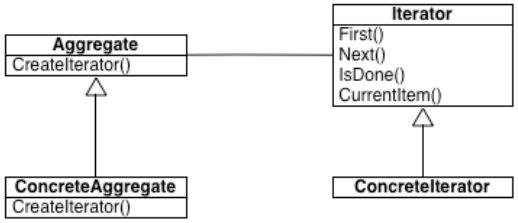
- Les patrons visent en général à accroître la qualité du code en visant un ou plusieurs des objectifs suivant:
  - Flexibilité accrue
  - Meilleure performance
  - Fiabilité accrue
- Attention! L'utilisation des patrons peut aussi augmenter la complexité du code
  - Par exemple, ajout d'indirections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

Types de patrons

- **Créationnels**: font l'abstraction du processus d'instanciation afin de rendre un système indépendant de la façon dont ses objets sont créés et représentés
- **Structurels**: se concentrent sur la façon dont les classes et les objets sont composés pour obtenir de plus grandes structures
- **Comportementaux**: décrivent les modèles de communication et interaction entre les objets

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



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();
}
```


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();
}
```


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 »

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)

Singleton - Exemple


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

    private Singleton () {}

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


Singleton - Exemple


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

    private LazySingleton() {}

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

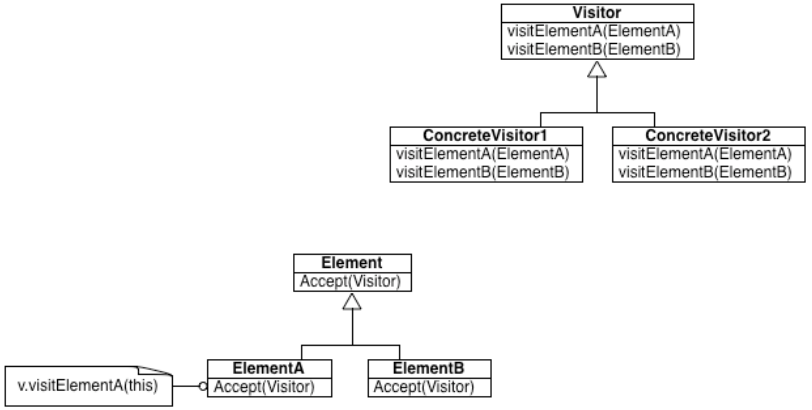

Singleton - Exemple


```
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;
    }
}
```

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)

Visitor – Diagramme de classes



Visitor - Exemple


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


Visitor - Exemple


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

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

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

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);
    }
}
```


Visitor – Exemple (alternative)

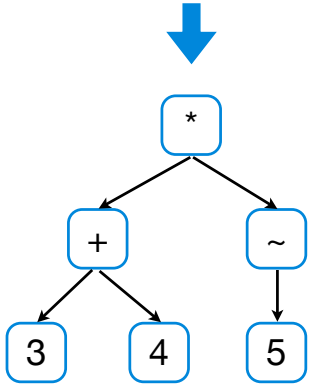

```
public class File {
    @Override
    public void accept(FSVisitor v) {
        v.visitFile(this);
    }
}

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);
    }
}
```


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);
    }
}
```


Visitor – Expressions

((3 + 4) * -5)



Visitor – Expressions


```
public interface Expression {
   public void accept(ExpressionVisitor v);
}

public class Number implements Expression {
   private int value;
   ...
}

public class BinaryOperation implements Expression {
   private Expression x;
   private Expression y;

   ...
}

public class UnaryMinus implements Expression {
   private Expression e;

   ...
}
```


Visitor – Expressions


```
public interface ExpressionVisitor {
   public void visitNumber(Number n);
   public void visitBinaryOperation(BinaryOperation op);
   public void visitUnaryMinus(UnaryMinus op);
}
```


Expressions - InfixPrinter


```
public class InfixPrinter implements ExpressionVisitor {
   public void visitNumber(Number n) {
      System.out.print(n);
   }

   public void visitBinaryOperation(BinaryOperation op) {
      System.out.print("(");
      op.getX().accept(this);

      System.out.print(" ");
      System.out.print(op.getKind());
      System.out.print(" ");

      op.getY().accept(this);
      System.out.print(")");
   }

   public void visitUnaryMinus(UnaryMinus op) {
      System.out.print("(-");
      op.getExpression().accept(this);
      System.out.print(")");
   }
}
```


Expressions - PostfixPrinter


```
public class PostfixPrinter implements ExpressionVisitor {
   public void visitNumber(Number n) {
      System.out.print(n);
      System.out.print(" ");
   }

   public void visitBinaryOperation(BinaryOperation op) {
      op.getX().accept(this);
      op.getY().accept(this);

      System.out.print(op.getKind());
      System.out.print(" ");
   }

   public void visitUnaryMinus(UnaryMinus op) {
      op.getExpression().accept(this);
      System.out.print("~ ");
   }
}
```

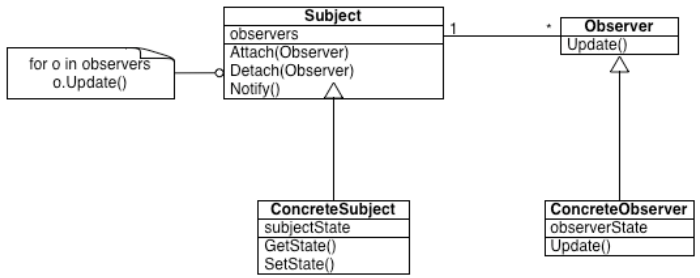
Observer (Comportemental)

- 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

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

Observer – Diagramme de classes



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();
          }
     }
}
```

Observer - Exemple

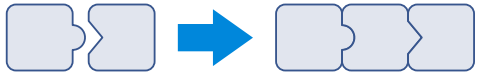

```
public class Timer extends Subject {
     public int getHours() { ... }
     public int getMinutes() { ... }
     public int getSeconds() { ... }

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


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 {...}
```

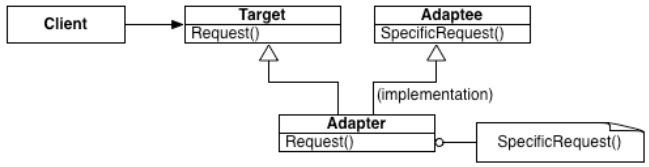

Adapter (Structurel)

- 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

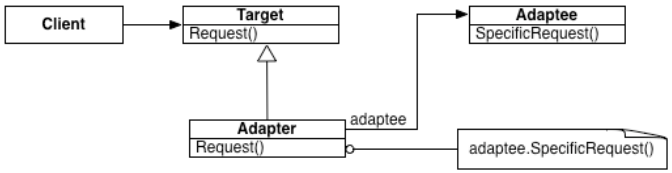


- Solutions
  - Par classe: héritage multiple / interfaces
  - Par objet: composition

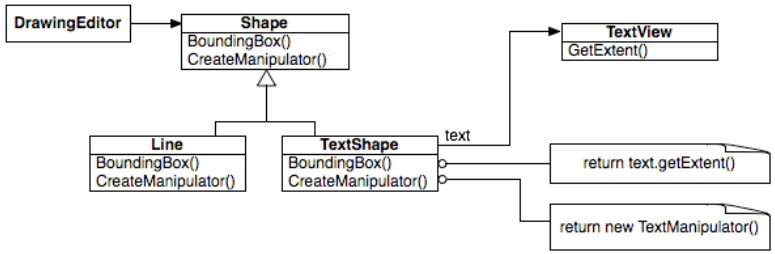
Adapter – Par classe



Adapter – Par objet



Adapter - Exemple



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();
    }
}
```