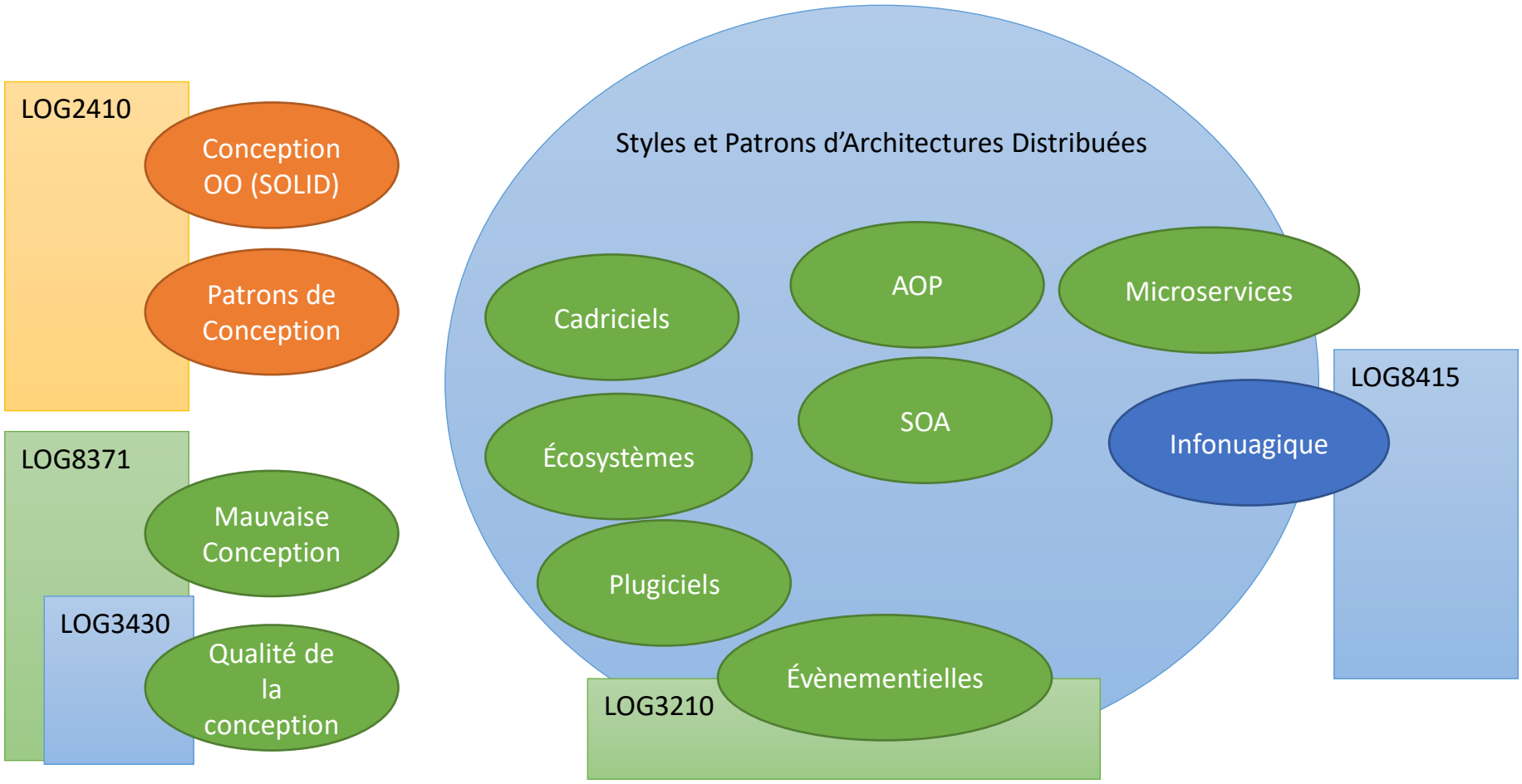
POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Chapitre 11

La Programmation Orientée Aspect (AOP)

Aujourd'hui


LOG2410
Conception OO (SOLID)
Patrons de Conception
LOG8371
Mauvaise Conception
LOG3430
Qualité de la conception
Styles et Patrons d'Architectures Distribuées
Cadriciels
AOP
Microservices
Écosystèmes
SOA
Infonuagique
Plugiciels
Évènementielles
LOG3210
LOG8415

Problème

- Vous développez un système bancaire en ligne. Le système devrait entre autres permettre à un utilisateur de transférer des fonds d'un compte à l'autre.

Solution #1

- Vous développez un système bancaire en ligne. Le système devrait entre autres permettre à un utilisateur de transférer des fonds d'un compte à l'autre.

```
void transfer(Account fromAcc, Account toAcc,
 int amount) throws Exception {
   if (fromAcc.getBalance() < amount)
       throw new InsufficientFundsException();

   fromAcc.withdraw(amount);
   toAcc.deposit(amount);
}
```

Plus de problèmes…

- Est-ce suffisant pour un système bancaire en ligne ?
- Qu'en est-il de la sécurité ?
  - Est-ce que n'importe qui peut transférer de l'argent entre deux comptes ?
- Qu'en est-il de l'intégrité ?
  - Peut-on toujours transférer de l'argent d'un compte ?
- Qu'en est-il de la transparence et de la traçabilité ?
  - Les transactions doivent être enregistrées et être récupérable pour fins de futures vérifications.

Solution #2

- Sécurité +
- Intégrité +
- Traçabilité +

```
void transfer(Account fromAcc, Account toAcc,
    int amount, User user,
    Logger logger, Database database) throws Exception {
  logger.info("Transferring money...");

  if (!isUserAuthorised(user, fromAcc)) {
    logger.info("User has no permission.");
    throw new UnauthorisedUserException();
  }

  if (fromAcc.getBalance() < amount) {
    logger.info("Insufficient funds.");
    throw new InsufficientFundsException();
  }

  fromAcc.withdraw(amount);
  toAcc.deposit(amount);

  database.commitChanges();  // Atomic operation.

  logger.info("Transaction successful.");
}
```

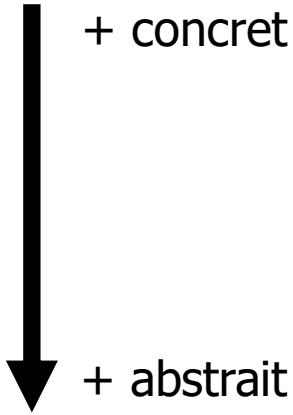
Avons-nous terminé?

- Supposons maintenant que ces enjeux (sécurité, intégrité, traçabilité) doivent être considérés dans chaque fonction du système bancaire.
  - Disons 10 méthodes…
  - Nous aurons beaucoup de répétitions…
- Imaginons maintenant que nous changions une partie de la politique de sécurité et que les transactions doivent dorénavant être enregistrées.
  - Nous devrons changer toutes les parties de code affectées.
  - Et si on en oublie une ?
  - Et si on ajoute du code ?
- Voyons si nous pouvons faire mieux…

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Les paradigmes de programmation

- Langages machine
- Langages procéduraux
- Langages Orienté Objet
- Langages Orienté Aspect



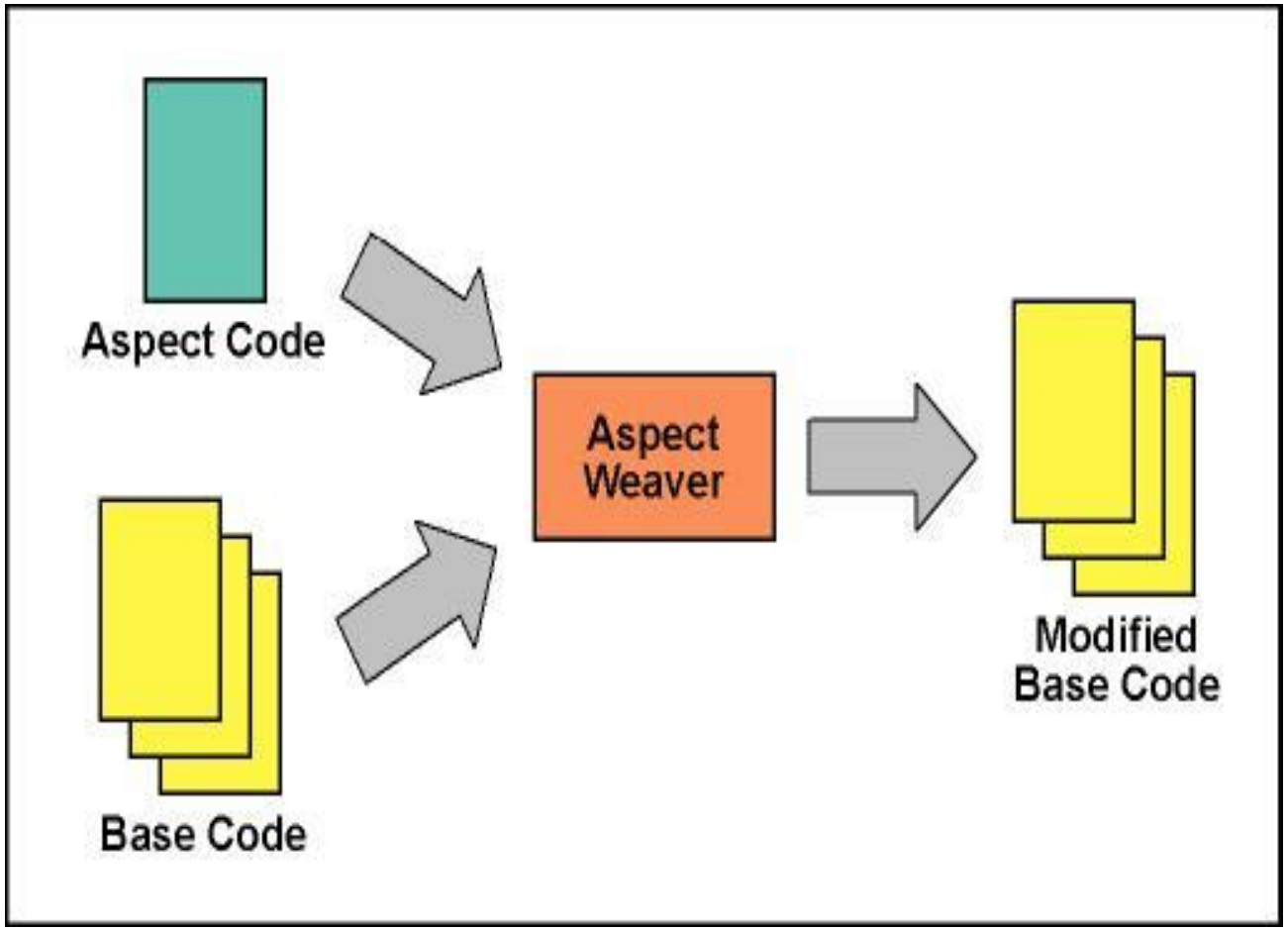
Avantages du paradigme OO

- Les langages OO sont très puissants et possèdent des avantages importants:
  - Réutilisation
  - Séparation interface/implantation
    - Généralisation, héritage, abstractions, encapsulation, polymorphisme, …
  - Typage
  - etc.

Problèmes du paradigme OO

- Mais, malgré tout, certains problèmes demeurent:
  - Difficulté à modéliser les fonctionnalités transversales («*crosscutting concerns*»).
    - Éparpillement de code (*code scattering*)
    - Confusion du code (*code tangling*)
  - Mène à des classes et/ou modules moins indépendants et donc moins réutilisables.
  - AOP fournit une solution pour modéliser les fonctionnalités transversales sans affecter les classes issues des concepts du domaine.

AOP: Principe général

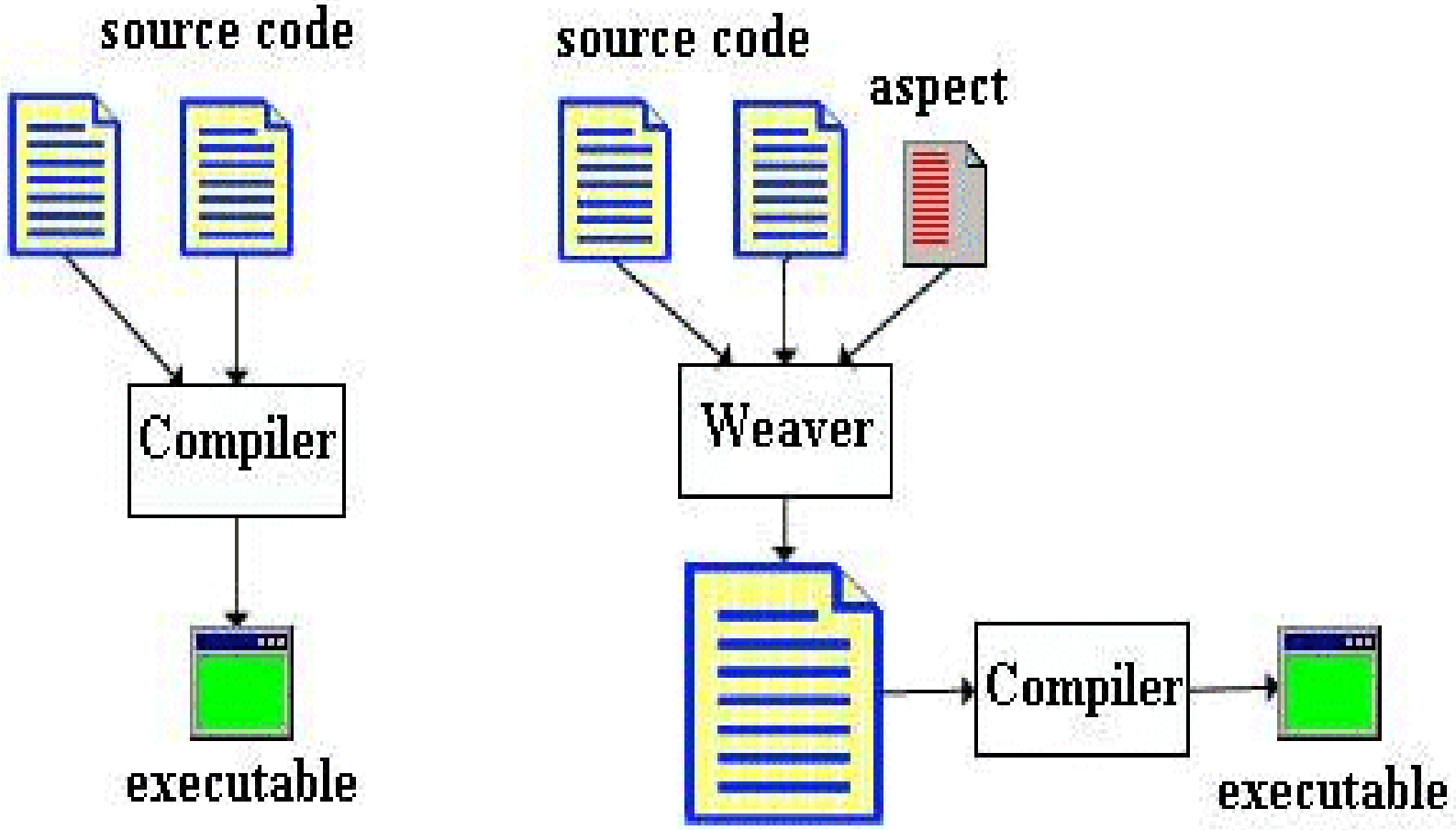


- http://control.ee.ethz.ch/~ceg/AutomatedFrameworkInstantiation/doc/AspectOrientedProgramming.html

AOP est combinée à des technologies existantes.

- On modélise les concepts issus du domaine du problème à résoudre avec les techniques habituelles (OO).
  - Note : En fait, AOP peut être utilisé non seulement en combinaison avec l'orienté objet, mais aussi avec des langages procéduraux comme le C.
- On modélise les fonctionnalités transversales (CCC) avec des aspects.
- Un processus nommé « *code weaving* » permet de convertir les aspects en code compilable par un compilateur OO standard.

Compilation standard et compilation avec AOP



- www.developer.com/design/article.php/3308941

Bénéfices

- Diminue l'éparpillement de code
  - Le code d'une fonctionnalité transversale est maintenant localisé à un endroit précis.
  - Rend plus facile la compréhension et la maintenance du code.
- Diminue la confusion du code
  - Les classes issues des concepts du domaine ont une plus forte cohésion. Elles ne sont pas (ou moins) «polluées» par du code issu de fonctionnalités transversales.
  - Ces classes sont donc plus réutilisables.

AOP: Fonctionnement

- On définit une nouvelle construction, nommée « **aspect** », qui permet de modéliser les fonctionnalités transversales.
- AOP permet 2 types de « *crosscutting* » sur un code.
  - « *Crosscutting* » dynamique,
  - « *Crosscutting* » statique.

AOP: « *Crosscutting* » dynamique

- Intercepte un évènement dans le déroulement du programme.
- Exécute des opérations avant/après/pendant cet évènement:
  - Après/Avant un appel de fonction.
  - Lors du lancement d'une exception.
  - Etc.
- Type le plus fréquent.

AOP: « *Crosscutting* » statique

- Modifie la structure statique des classes
  - Ajout d'attributs ou de fonctions à une classe
  - Modification de la hiérarchie des classes
  - Ajout de contraintes à la compilation
- Est utilisé entre autres pour implanter le «*crosscutting*» dynamique.

Un exemple simple avec AspectJ

```
public class HelloWorld
{
    public void hello()
    {
        System.out.println( "Hello World" );
    }
}

public class Main
{
    public static void main( String[] args )
    {
        HelloWorld h = new HelloWorld();
        h.hello();
    }
}
```

En compilant et exécutant la classe **Main**, on obtient...

```
[selafa@d6128-02 bin]$ java Main
Hello World
```

Un exemple simple avec AspectJ

```
public aspect AspectHelloWorld
{
    protected pointcut helloWorldPointcut():
        call( * HelloWorld.*(..) );

    before(): helloWorldPointcut()
    {
        System.out.print( "Mon premier aspect... " );
    }
}
```

En compilant et exécutant la classe **Main** en ajoutant l'aspect **AspectHelloWorld**, on obtient...

```
[selafa@d6128-02 bin]$ java Main
Mon premier aspect... Hello World
```

AspectJ: la notion d'aspect

- Un aspect est une construction semblable à une classe permettant d'implanter des fonctionnalités transversales.
- Similarités avec une classe:
  - Un aspect peut inclure des attributs et des méthodes.
  - Les même règles de visibilité (public, private, protected, packaged).
  - Un aspect peut être abstrait.
  - Un aspect peut dériver d'une classe, d'une interface ou d'un autre aspect (abstrait).
  - On peut faire des aspects embarqués tout comme avec les classes (nested aspect).

AspectJ: la notion d'aspect

- Un aspect est toutefois très différent d'une classe à plusieurs niveaux:
  - On ne peut pas directement construire une instance d'un aspect.
  - On ne peut pas faire dériver un aspect d'un aspect concret.
  - Un aspect peut être en mode « privileged » et accéder aux éléments privés des classes qu'il affecte.

AspectJ: *Crosscutting* dynamique

- *Pointcuts*
  - Les *pointcuts* permettent de capturer un(des) point(s) d'exécution (« *join point* ») dans le déroulement d'un programme.
    - `protected pointcut nomPointcut():call( * Toto.*(..) );`
- *Advices*
  - Les *advices* permettent d'exécuter du code lorsqu'un *pointcut* est actif. On peut exécuter du code avant ou après l'activation du *pointcut* ou encore modifier l'environnement d'exécution du *pointcut*.
    - `before(): nomPointcut() {...}`
    - `after(): nomPointcut() {...}`
    - `Object around(): nomPointcut() {...};`

AspectJ: *Crosscutting* statique

- Introduction de membres
  - Le *crosscutting* statique permet d'introduire des membres à une classe.

```
public aspect UnAspect
{

   private float UneClasse.x; // Ajoute un attribut à la
                              // classe UneClasse

   public float UneClasse.getX() // Ajoute une méthode à la
   {                             // classe UneClasse
      return x;
   }

   ...
}
```

AspectJ: *Crosscutting* statique

- Modifier la hiérarchie des classes
  - On peut également modifier la hiérarchie des classes.

```
public aspect UnAspect
{
   // Déclare une interface interne à l'aspect
   protected interface UneInterface{};

   // On décide tyranniquement que la classe UneClasse doit
   // implanter cette interface
   declare parents: UneClasse implements UneInterface;

   ...
}
```

AspectJ: *Crosscutting* statique

- Modifier le comportement du compilateur
  - Le *crosscutting* statique permet également de définir des messages à la compilation

```
public aspect UnAspect
{
   protected pointcut pointcut1()
      : call( void UneClasse.fctDesuete1() );
   protected pointcut pointcut2()
      : call( void UneClasse.fctDesuete2() );

   declare error: pointcut1()
      : "Fonction désuète... utilisez plutôt UneClasse.nvlleFct1()"

   declare warning: pointcut2()
      : "Fonction désuète... Utilisez plutôt UneClasse.nvlleFct2()"

   ...
}
```

L'exemple classique: trace d'exécution

- L'exemple le plus connu d'utilisation de AOP est sans aucun doute celui du « *logfile* ».
- Supposons que l'on veuille garder une trace d'exécution de notre programme.
  - La technique habituelle consiste à ajouter du code dans chacune des fonctions du système pour faire des entrées dans l'output choisi.
  - Ceci est long, ardu, confus, et très propice aux erreurs de toutes sortes...

Trace d'exécution

- La méthode classique est très invasive et propice aux erreurs…

```
public class Toto {
   public void fct1() {
      Output.ajouterEntree( "entre dans la fct fct1()" );
      ...
      Output.ajouterEntree( "sors de la fct fct1()" );
   }
   public void fct2() {
      Output.ajouterEntree( "entre dans la fct fct2()" );
      ...
      Output.ajouterEntree( "sors de la fct fct2()" );
   }
   ...
}
```

- Beurk !

Trace d'exécution

- Avec AOP, quelques lignes claires et simples placées dans un aspect permettront d'enregistrer la trace d'exécution de manière non invasive.
  - Il n'est plus nécessaire de modifier chacune des fonctions indépendamment.
  - Il est facile de modifier la méthode d'enregistrement ou le format des entrées puisque le code associé à la fonctionnalité est entièrement localisé dans un aspect.
- C'est tout simplement GÉNIAL, non ?

Trace d'exécution: CompteBancaire

```
public class CompteBancaire
{
   private double tauxInteret = 0;
   private double solde = 0;
   private String client = "";

   public CompteBancaire( double tauxInteret,
                          double solde,
                          String client )
   {
      super();
      this.tauxInteret = tauxInteret;
      this.solde = solde;
      this.client = client;
   }

   public void crediter( double montant )
   {
      this.solde += montant;
   }

   // à suivre ...
```

Trace d'exécution: CompteBancaire

```
   // suite ...

   public void debiter( double montant )
                                  throws ExceptionFondsInsuffisants
   {
      if ( montant > this.solde )
      {
         throw new ExceptionFondsInsuffisants( this.solde, montant  );
      }
      this.solde -= montant;
   }

   public void mettreAJourSolde()
   {
      this.solde *= 1.0 + this.tauxInteret / 100.0;
   }

   [...]

} // Fin de la classe CompteBancaire
```

Trace d'exécution: ExceptionFondsInsuffisants

```
public class ExceptionFondsInsuffisants extends Exception
{
   public ExceptionFondsInsuffisants( double solde,
                                      double montant )
   {
      super(   "Il n'y a pas assez de fonds ("
             + solde
             + ") pour debiter le montant voulu ("
             + montant
             + ")" );
   }
} // Fin de la classe ExceptionFondsInsuffisants
```

Trace d'exécution: classe principale

```
public class SystemeBancaire
{

   public static void main( String[] args )
                          throws ExceptionFondsInsuffisants
   {
      CompteBancaire c1 = new CompteBancaire( 2.25, 1000.0,
                                              "James Bond" );
      CompteBancaire c2 = new CompteBancaire( 3.25, 2000.0,
                                              "Darth Vader" );

      c1.crediter( 100 );
      c2.debiter( 500 );
      c1.mettreAJourSolde();
   }
} // Fin de la classe SystemeBancaire
```

- Maintenant, si on veut avoir une trace d'exécution du système bancaire, il suffit de créer l'aspect approprié.

Trace d'exécution : AspectTraceDExecution

```
public aspect AspectTraceDExecution {
   private int profondeur = -1;

   private pointcut pointsATracer(): !within( AspectTraceDExecution );

   before(): pointsATracer() {
      profondeur++;
      print( "Before", thisJoinPoint );
   }

   after(): pointsATracer() {
      print( "After", thisJoinPoint );
      profondeur--;
   }

   private void print( String prefix, Object message ) {
      for ( int i=0; i<(profondeur*2); i++ ) {
         System.out.print( " " );
      }
      System.out.println( prefix + ": " + message );
   }
} // Fin de l'aspect AspectTraceDExecution
```

Trace d'exécution: output

```
Before: staticinitialization(Main.<clinit>)
After: staticinitialization(Main.<clinit>)
Before: execution(void Main.main(String[]))
  Before: call(CompteBancaire(double, double, String))
    Before: staticinitialization(CompteBancaire.<clinit>)
    After: staticinitialization(CompteBancaire.<clinit>)
    Before: preinitialization(CompteBancaire(double, double, String))
    After: preinitialization(CompteBancaire(double, double, String))
    Before: initialization(CompteBancaire(double, double, String))
      Before: execution(CompteBancaire(double, double, String))
        Before: set(double CompteBancaire.tauxInteret)
        After: set(double CompteBancaire.tauxInteret)
        Before: set(double CompteBancaire.solde)
        After: set(double CompteBancaire.solde)
[...]
  Before: call(void CompteBancaire.crediter(double))
    Before: execution(void CompteBancaire.crediter(double))
      Before: get(double CompteBancaire.solde)
      After: get(double CompteBancaire.solde)
      Before: set(double CompteBancaire.solde)
      After: set(double CompteBancaire.solde)
    After: execution(void CompteBancaire.crediter(double))
  After: call(void CompteBancaire.crediter(double))
[...]
```

AOP: Mythes et réalités

- *Le flot de contrôle du programme est difficile à suivre*.
  - Effectivement. Mais c'est ce qu'on souhaite ! On veut travailler à un niveau d'abstraction plus élevé.
  - Et le polymorphisme ?
- *AOP ne règle pas de nouveaux problèmes*.
  - En effet. On ne cherche pas à résoudre de nouveaux problèmes, mais à résoudre les problèmes actuels de façons plus efficace.
- *AOP encourage les mauvais designs*.
  - Faux. AOP nous permet de corriger des problèmes difficilement corrigibles en OO, mais la bonne conception OO des classes issues du domaine du problème reste nécessaire.
- *AOP peut être utile, mais la définition correcte d'interfaces est tout aussi efficace*
  - Faux. Pensez à notre exemple de « *logfile* »: l'utilisation d'une librairie pour enregistrer les traces d'exécution nécessite quand même une invocation dans chaque fonction dont on veut garder la trace.

AOP: mythes et réalités

- *AOP ne fait que « patcher » la conception des classes du domaine*.
  - Faux. AOP permet d'augmenter la modularité, la compréhensibilité et la traçabilité du code.
- *AOP brise l'encapsulation*.
  - Vrai, mais d'une façon systématique et contrôlée. On peut briser l'encapsulation d'une classe à partir d'un aspect. AOP est donc très puissant, mais ce pouvoir doit être utilisé correctement...
- *AOP va remplacer l'OO*.
  - Faux. Les concepts issus du domaine du problème continuent d'être modélisés avec des objets. AOP est utile pour modéliser les fonctions transversales qui, autrement, nécessiteraient des modifications à plusieurs classes du domaine.
- *AOP n'est pas mature*.
  - AspectJ en est à la version 1.9.6
    - Il est parfaitement intégré à plusieurs IDE: Eclipse, NetBeans, JBuilder, Emacs JDDE.
  - AspectC++ 2.2 est disponible.
    - Il est intégré à Eclipse.
  - AOP est disponible dans un grand nombre de langages:
    - Java, C#, C/C++, PHP, Haskell, JavaScript, Matlab, Perl, Prolog, Python, Ruby, XML

AOP : Un petit résumé

- AOP permet de modéliser des fonctionnalités qui, autrement, seraient invasives.
  - Fonctionnalités transversales ou *crosscutting concerns*.
- AOP ne vise pas à remplacer les langages OO, mais à les compléter.
  - Les concepts issus du domaine sont toujours modélisés avec des classes
  - Les fonctionnalités transversales sont modélisées avec des aspects.
  - Réduit la confusion et l'éparpillement du code (*code tangling* et *code scattering*).

Champs d'application

- Les champs d'application d'AOP sont très variés. La plupart des applications peuvent bénéficier d'AOP à différents degrés.
- Aussitôt qu'une fonctionnalité transversale fait son apparition, AOP peut être utile.

Champs d'application: Les design patterns

- Un certains nombre de travaux tendent à démontrer que l'implantation de certains patrons de conception avec AOP est avantageuse…
- Hannemann et Kiczales[1,2] ont implanté les 23 patrons du Gang of Four[3].
- Il sont arrivés aux résultats suivants:
  - Meilleure localité (17/23)
  - Possibilité de réutilisation (12/23)
  - Transparence à la composition de patrons (14/23)
  - Facilité à (dés)instancier un patron (17/23)

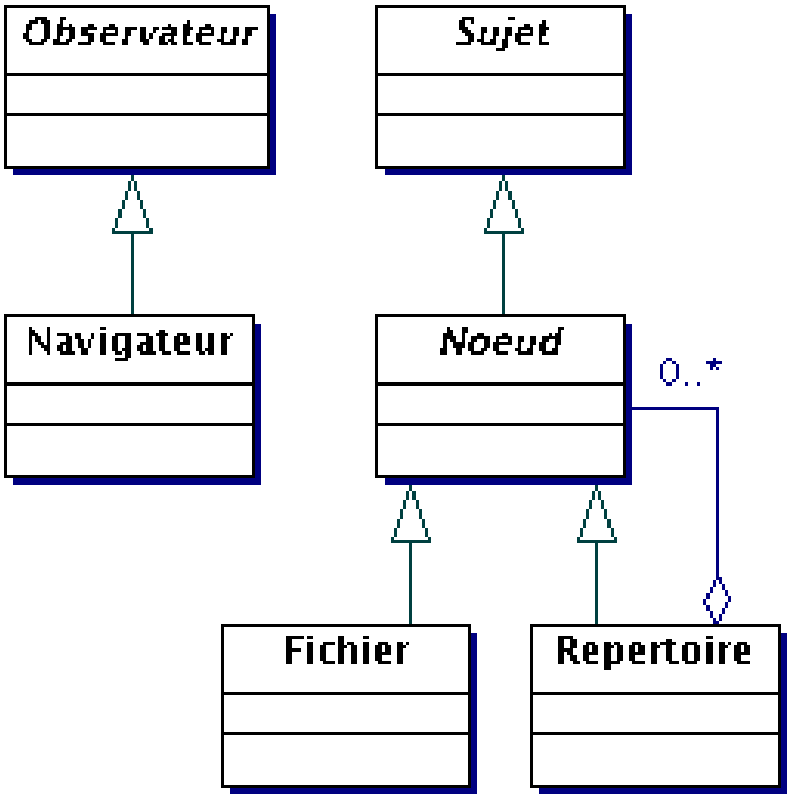
[1]: Hannemann and Kiczales. *Design Pattern Implementation in Java and AspectJ*, 17th Annual ACM conference on OOPSLA, pages 161-173, November 2002.

[2]: http://www.cs.ubc.ca/~jan/AODPs/

[3]: E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., 1995.

Exemple : Le patron observer

- La solution orientée objet classique :



Cette solution est invasive:

- Le code associé au patron est **éparpillé** dans plusieurs classes différentes (code scattering).
- Il faut modifier les classes issues du domaine, `Navigateur` et `Noeud` pour les faire participer au patron
- Lorsqu'on veut appliquer ou retirer le patron, on doit modifier la hiérarchie statique des classes existantes.
- Les classes existantes ont perdu de la cohésion (code tangling) et sont moins réutilisables.

Exemple : Le patron observer

- Avec AOP, on peut définir le patron dans un aspect et les classes issues du domaine comme **`Noeud`** et **`Navigateur`** ne seront pas affectées.
- En fait, on crée deux aspects
  - Un aspect abstrait définissant les éléments communs à toutes les implantations du patron Observer.
  - Un aspect concret qui concrétise l'utilisation du patron dans un contexte particulier.
- Le patron sera appliqué de l'extérieur.

Exemple : Le patron observer

```
public abstract aspect ObserverProtocol
{
   protected interface Subject {};

   protected interface Observer {};

   private WeakHashMap perSubjectObservers;

   protected List getObservers( Subject s )
   {
      if ( perSubjectObservers == null ) {
         perSubjectObservers = new WeakHashMap();
      }
      List observers = (List)( perSubjectObservers.get(s) );
      if ( observers == null ) {
         observers = new LinkedList();
         perSubjectObservers.put( s, observers );
      }
      return observers;
   }

   // à suivre ...
```

Exemple : Le patron observer

```
   // ... Suite de l'aspect ObserverProtocol

   public void addObserver( Subject s, Observer o )
   {
      getObservers(s).add(o);
   }

   public void removeObserver( Subject s, Observer o )
   {
      getObservers(s).remove(o);
   }

   abstract protected pointcut subjectChange( Subject s );

   abstract protected void updateObserver( Subject s, Observer o );

   after( Subject s ): subjectChange( s )
   {
      Iterator iter = getObservers(s).iterator();
      while ( iter.hasNext() ) {
         updateObserver( s, (Observer)iter.next() );
      }
   }
} // Fin de l'aspect ObserverProtocol
```

Exemple : Le patron observer

```
public aspect ObservateurNoeud extends ObserverProtocol
{
   declare parents: Navigateur implements Observer
   declare parents: Noeud implements Subject

   protected pointcut subjectChange( Subject s ) :
      (    call( void Noeud.adopter(Noeud) )
        || call( void Noeud.abandonner(Noeud) ) )
        && target( s );

   protected void updateObserver( Subject s, Observer o )
   {
      Navigateur n = (Navigateur) o;
      n.mettreAJour( (Noeud) s );
   }
}
```

Avantages de l'implantation avec AOP

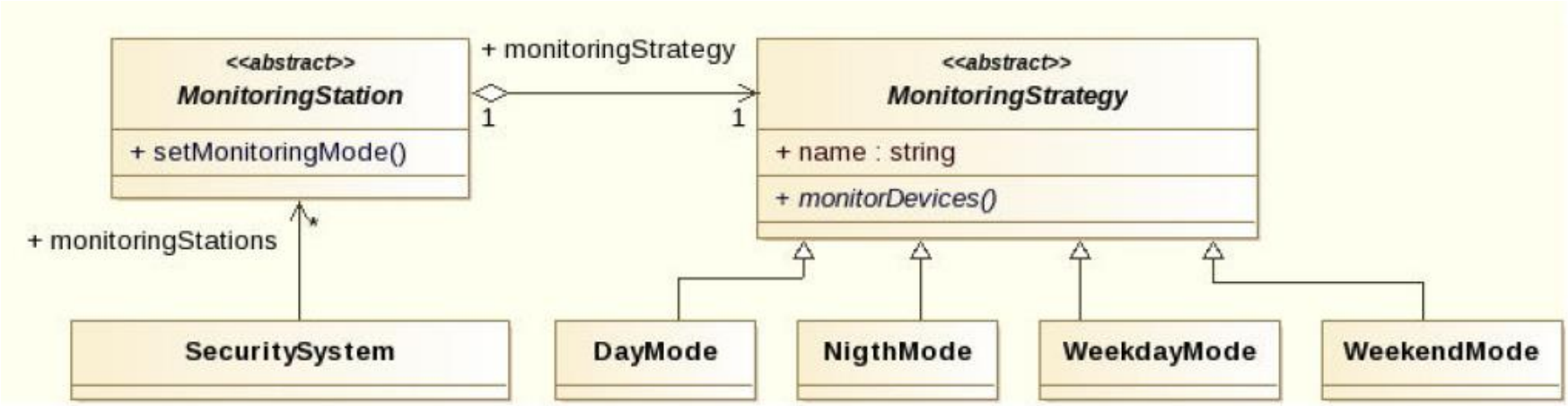
- Le code associé à l'implantation du patron est localisé à un endroit bien identifié.
  → Diminue l'éparpillement de code
- Les classes **`Noeud`** et **`Navigateur`** ne sont pas affectées par l'instanciation du patron.
  → Augmente la cohésion des classes issues du domaine.
  → Facilite l'instanciation et la « *désinstanciation* » du patron.
- Transparence à la composition de patrons.
  → Les classes **`Noeud`** et **`Navigateur`** pourraient participer simultanément à plusieurs patrons de façon complètement transparente.
- On peut définir une librairie de patrons de conception réutilisable en regroupant les aspects communs à toutes les instanciations des patrons dans des aspects abstraits.

Exercice

- On veut implémenter un système de sécurité qui utilise plusieurs dispositifs (cameras, détecteurs de mouvement, serrures) pour surveiller et assurer la sécurité d'un building. Selon la journée (jour de semaine ou de fin de semaine) et l'heure (jour vs. nuit), le système utilise différents dispositifs (e.g. la nuit uniquement les caméras).
- Quel patron de conception utiliseriez-vous pour implémenter ce système ?
- Comment peut-on transformer cette conception en utilisant l'AOP ?

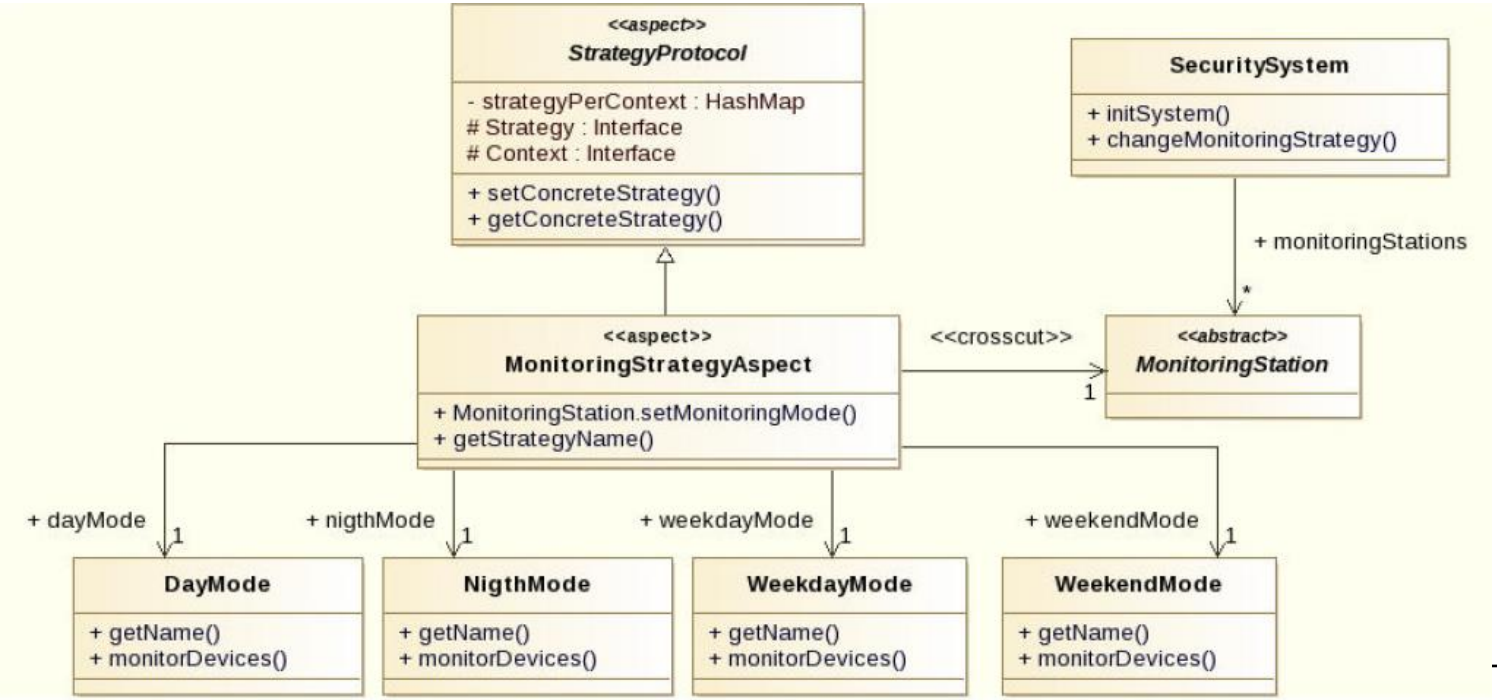
Patrons OO: Strategy

- Chaque stratégie concrète réduit les dispositifs à surveiller à un sous-ensemble de l'ensemble original.
- En cours d'exécution on vérifie l'heure et on choisit la stratégie appropriée.



Solution AOP

- La stratégie abstraite est remplacée par un aspect abstrait StrategyProtocol, qui conserve une paire de Contexte (temps) et Stratégie et un aspect.
- L'AOP réduit la profondeur de l'héritage and libère les stratégies concrètes, comme classes du domaine, pour étendre d'autres classes.



Impact de la solution AOP

Réduit

- L'enchevêtrement du code
- La dispersion du code
- La profondeur de l'héritage
- La compléxité

Accroît

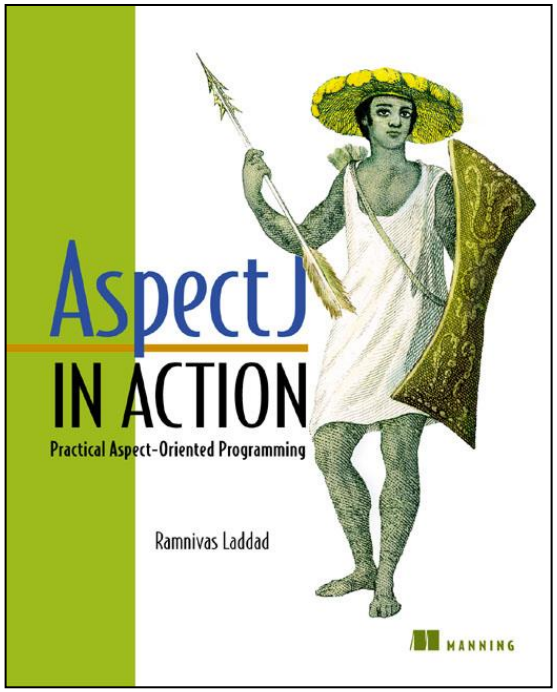
- Le couplage
- Cohésion
- NLC
  - Parce que la stratégie est enchevêtrée avec le patron Observer.

AOP : Sources de documentation

- Information générale
  - Le site web du Aspect Oriented Software Development Community (AOSD) : aosd.net.
  - Le site web de AspectJ, une implantation Java de AOP, faites dans le cadre du projet Eclipse : eclipse.org/aspectj.
  - Le site de AspectC++, une implantation C++ de AOP : www.aspectc.org.
- Quelques travaux
  - Une librairie de patrons implantée en AspectJ : www.cs.ubc.ca/~jan/AODPs/.
  - Article sur les problèmes des implantations OO des patrons de conception : www-lsr.imag.fr/OOIS_Reuse_Workshop/Papers/Hachani.pdf.
- Google !!!

AOP : Sources de documentation

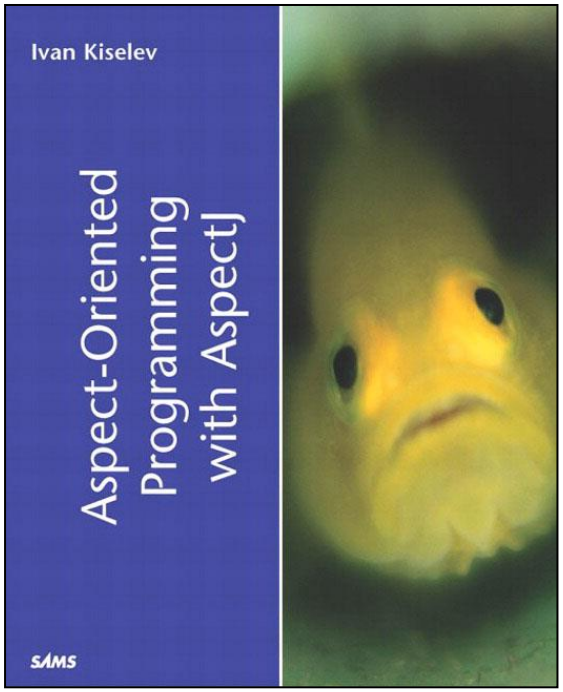
- Quelques livres intéressants :



•**AspectJ in Action**

•Ramnivas Laddad

•Manning, 2003



•**Aspect-Oriented Programming with AspectJ**

•Ivan Kiselev

•SAMS, 2002

Prochaine fois