Concepts et applications de l'AOP

Première partie : Introduction, motivations et limites de l'AOP

Félix-Antoine Bourbonnais

Université Laval, Québec

25 juin 2009

Plan

Gestion de dépendances

Lors d'une modification ou d'un ajout, on veut :

- Limiter les modules à modifier directement
- Limiter les modules impactés indirectement
- Concentrer les modifications dans un groupe logique et prévisible

Solutions

1. Architecture logicielle : gestion des dépendances (prévention)
   - Principes et lois OO (Liskov, Demeter, Open-Closed, etc.)
   - Bonnes pratiques et principes architecturaux (patrons, etc.)
   - AOP !
   - ...
2. Les tests (vérification)

Couplage et cohésions

Définition

Couplage : Degré de dépendance entre deux modules.

Objectif : Couplage FAIBLE

Définition

Cohésion : Degré de collaboration fonctionnelle entre les éléments d'un même module pour répondre à la préoccupation (but) de ce module.

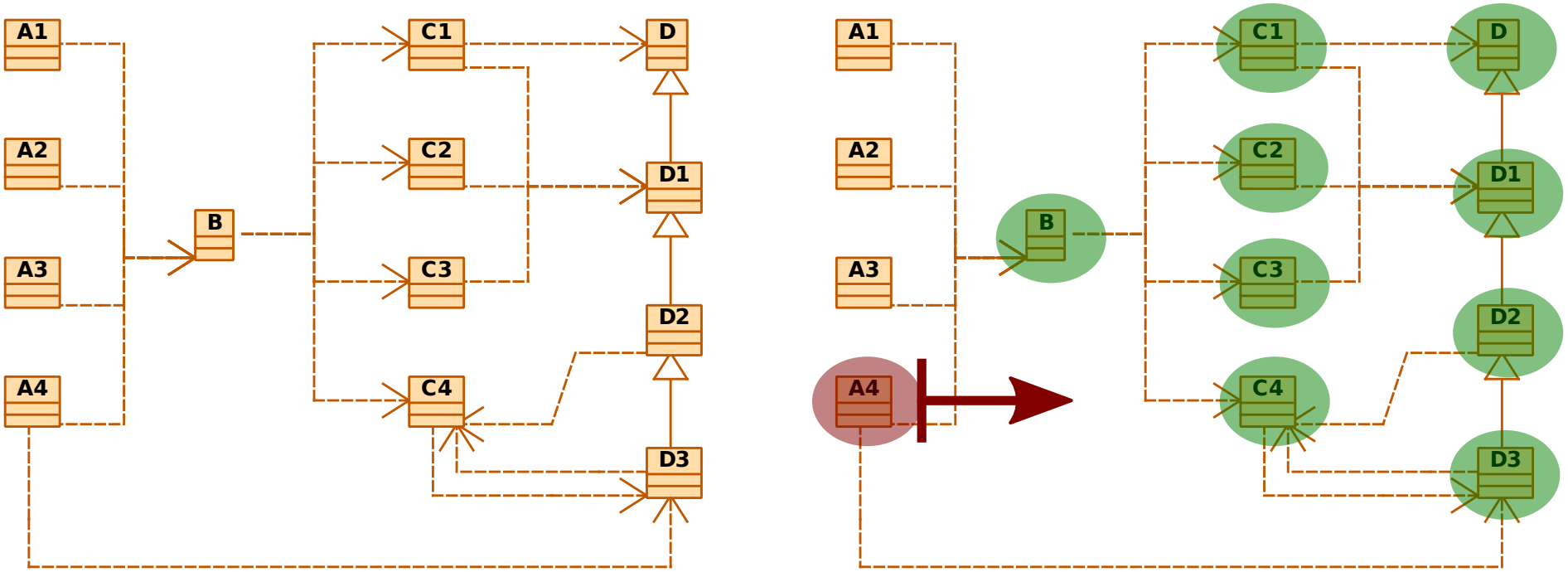
Objectif : Cohésion FORTE

1. Les constituants d'une classe devraient être fortement liés entre eux
2. La relation entre les classes d'un module devrait être plus forte que le lien avec les classes d'autres modules.

Couplage faible

- La modification d'une fonctionnalité est isolée
  - Éviter l'effet d'avalanche (« ripple effect »)
- Il est plus facile de comprendre et modifier un module qui n'a pas trop de dépendances
- L'élément peut être réutilisé dans un autre contexte/projet
  - Sans devoir importer les dépendances, puis les dépendances des dépendances, ...
- L'élément peut être testé séparément du système (isolation)

Couplage faible (suite)



Plan

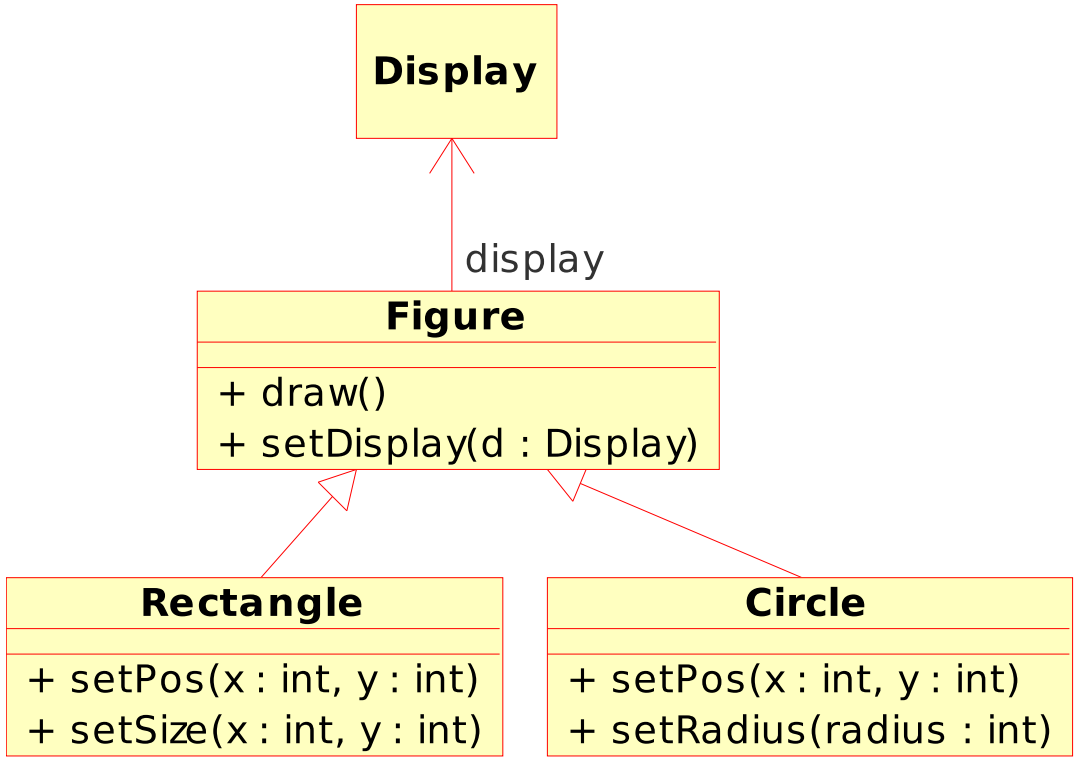
Plan

POO simple

```
class Figure {
    private Display display; /* <----- */
    void draw();
    void setDisplay(Display display);
}

class Rectangle extends Figure {
    private int x1, x2, y1, y2;
    void setPos(int x1, int y1) {
        this.x1 = x1; this.y1 = y1;
        display.update(); /* <---- */
    }
    void setSize(int x2, int y2) {
        this.x2 = x2; this.y2 = y2;
        display.update(); /* <---- */
    }
}

class Circle extends Figure {
    private int x, y, radius;
    void setPos(int x, int y) {
        this.x = x; this.y = y;
        display.update(); /* <---- */
    }
    void setRadius(int radius) {
        this.radius = radius;
        display.update(); /* <---- */
    }
}
```

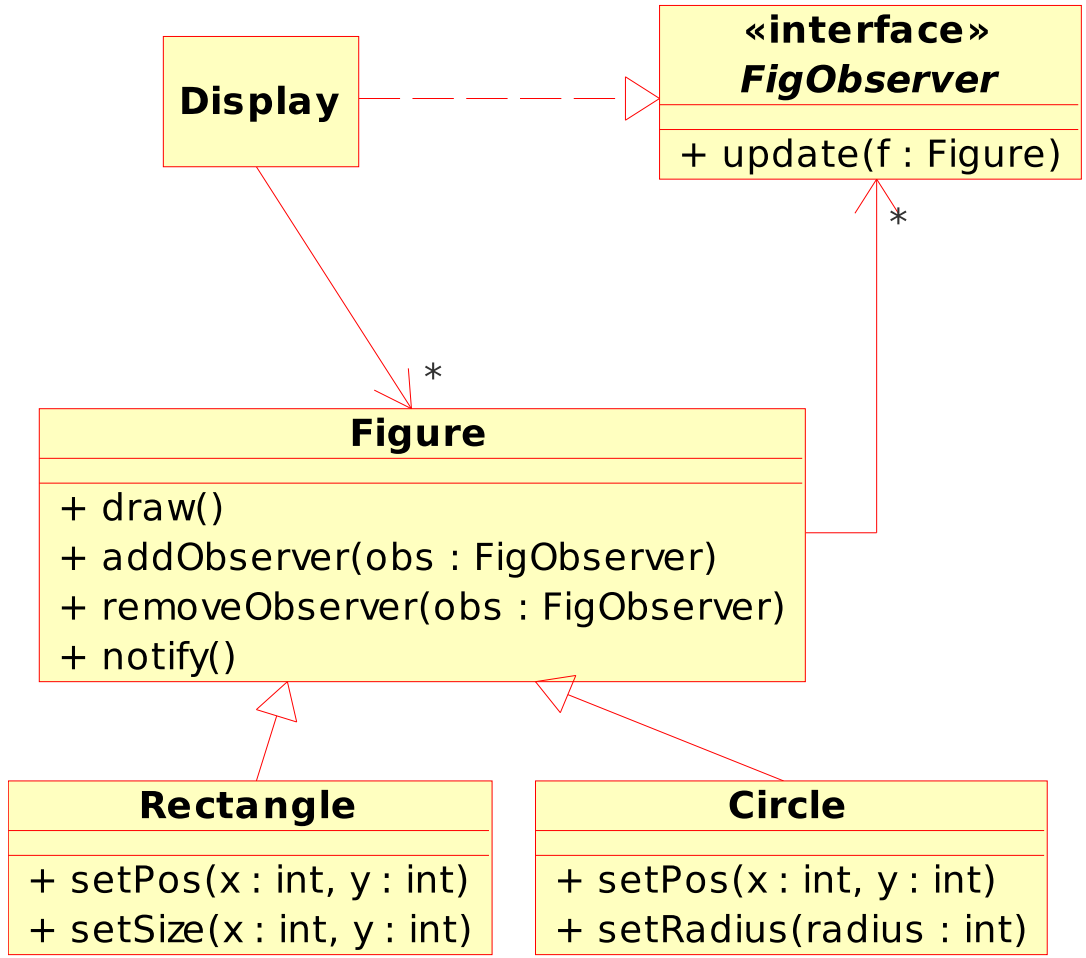


POO + Patterns

```
interface FigObserver { /* <--- */
    void update(Figure f);
}

class Figure {
    private List<FigObserver> obs;
    void addObserver(FigObserver o);/* <--- */
    void removeObserver(FigObserver o);/*<--*/
    void notify(); /* <--- */
    void draw();
}

class Rectangle extends Figure {
     private int x1, x2, y1, y2;
     void setPos(int x1, int y1) {
          this.x1 = x1; this.y1 = y1;
          notify(); /* <----- */
     }
     void setSize(int x2, int y2) {
          this.x2 = x2; this.y2 = y2;
          notify(); /* <----- */
     }
}
```



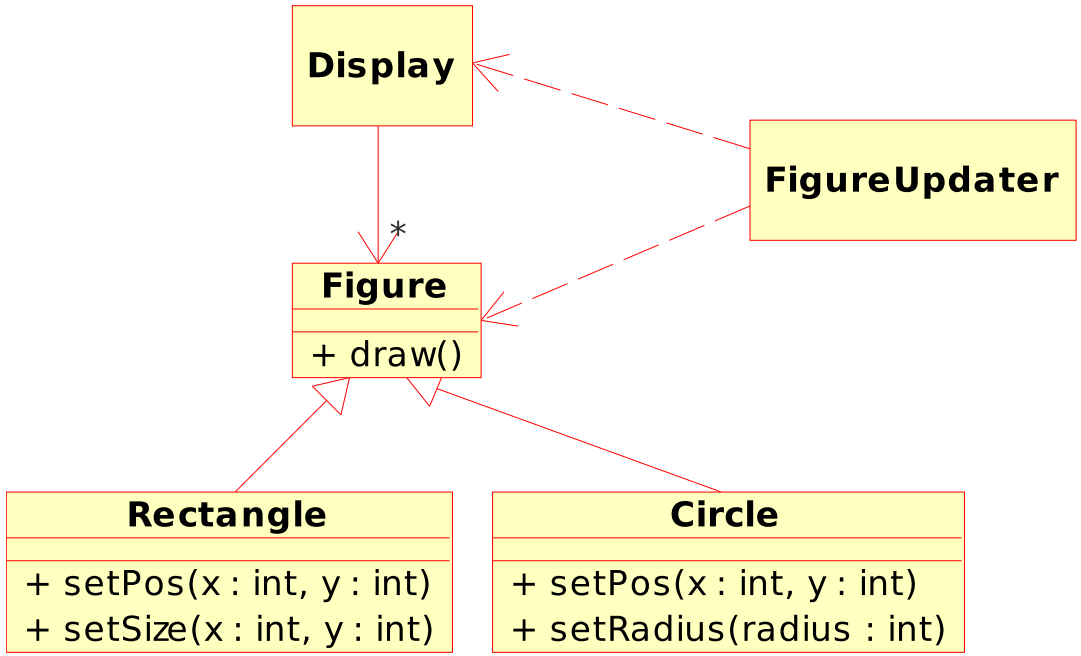
Avec aspects

```
aspect FigureUpdater {
    poincut figChanged(Figure f) :
        target(f) &&
        call(* set*(..));

    after(Figure f) returning : figChanged(f) {
        Display.update(f);
    }
}

class Figure {
    void draw();
}

class Rectangle extends Figure {
    private int x1, x2, y1, y2;
    void setPos(int x1, int y1) {
        this.x1 = x1; this.y1 = y1;
    }
    void setSize(int x2, int y2) {
        this.x2 = x2; this.y2 = y2;
    }
}
```



Types de préoccupations

Un logiciel est constitué de plusieurs préoccupations (concepts ou champs d'intérêts) :

1. Primaires (noyaux, critiques) :
   - Répondent aux besoins fondamentaux du logiciel (but du logiciel)
   - Préoccupations fonctionnelles (« business logic »)
2. Secondaires (systèmes, utilitaires)

Préoccupations dans un logiciel bancaire

- Gestion des comptes [Primaire]
- Gestion des transactions bancaires [Primaire]
- Journalisation (logging) [Secondaire]
- Sécurité [Secondaire]
- Authentification [Secondaire]
- Intégrité des transactions [Secondaire]

Préoccupations transverses

Crosscutting concerns

Définition

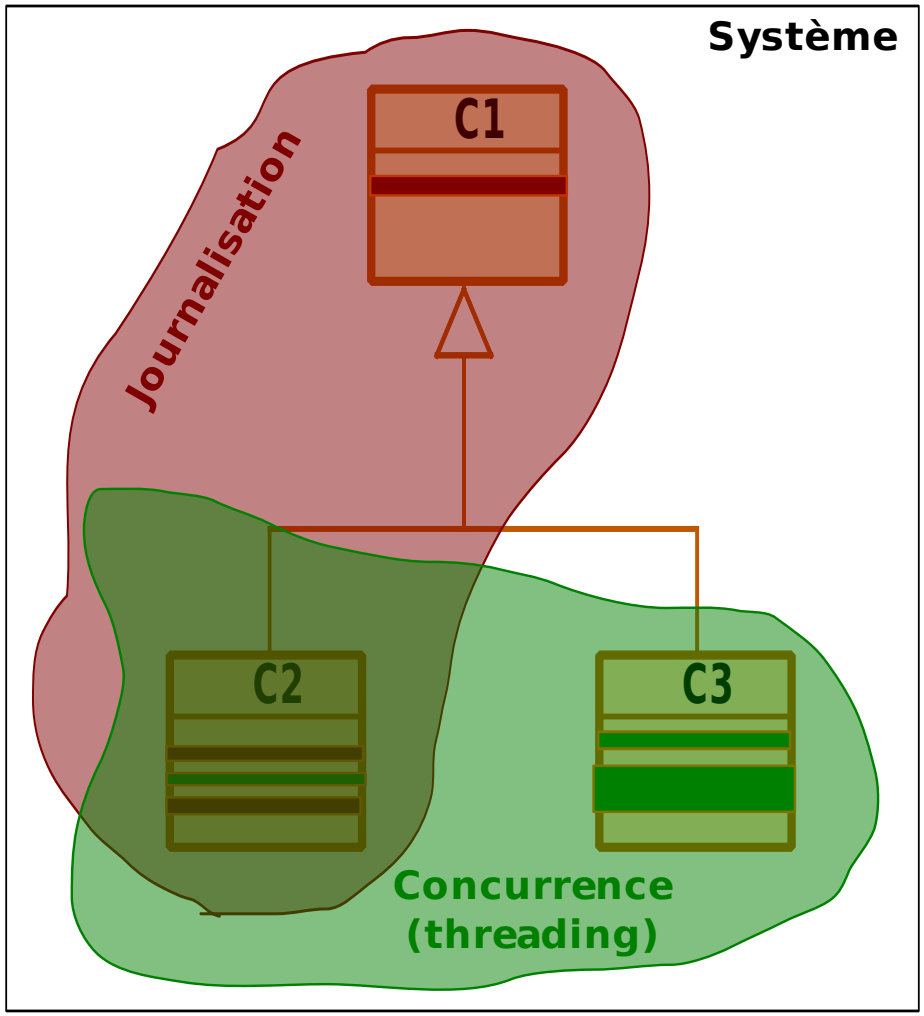
Les préoccupations secondaires (systèmes) ont tendance à contaminer plusieurs modules. Ces préoccupations sont alors appelées *préoccupations transverses (croscutting concerns)* car ils recoupent plusieurs modules sans pour autant contribuer à la préoccupation principale.

Fait

Un module destiné à encapsuler une préoccupation se trouve contaminé par du code destiné à d'autres préoccupations secondaires.

Préoccupations transverses

Crosscutting concerns



Exemple

```
1  /**
2    *Composer le numéro de téléphone demandé
3    */
4  public void composerTelephone(String numero) {
5      Telephone phone =Telephone.getinstance();
6
7      DAO dao =DAO.getCurrentDAO();
8      boolean db_is_locked =false;
9      DBTransaction transaction =null;
10     try {
11         dao.doRequest(new Request.SelectForUpdate("phone_table"));
12         db_is_locked =true;
13         transaction =dao.beginTransaction();
14
15         try {
16             telephone.composer(numero)
17         } catch (AccesRefuse e) {
18             // On n'avait pas le droit d'utiliser le téléphone
19             // (vérification d'accès a échoué)
20             Logger.getLogger().log(LOG_ERROR, "Acces refuse");
21             throws new RedirectToLoginPage();
22         }
23
24      transaction.commit();
25     } catch (Exception e) {
26         Logger.getLogger().log(LOG_ERROR, "Impossible de composer");
27         if(!db_is_locked) {
28             return;
29         }
30         Logger.getLogger().log(LOG_DEBUG, "Liberation de la table");
31         dao.unlockTable("phone_table");
32
33         if(transaction !=null) {
34             Logger.getLogger().log(LOG_DEBUG, "Renversement de la BD OK");
35             transaction.rollback();
36         }
37         return;
38     }
```

Exemple

Préoccupation transverse:
Base de données

```
1  /**
2    *Composer le numéro de téléphone demandé
3    */
4  public void composerTelephone(String numero) {
5      Telephone phone =Telephone.getinstance();
6
7      DAO dao =DAO.getCurrentDAO();
8      boolean db_is_locked =false;
9      DBTransaction transaction =null;
10     try {
11         dao.doRequest(new Request.SelectForUpdate("phone_table"));
12         db_is_locked =true;
13         transaction =dao.beginTransaction();
14
15         try {
16             telephone.composer(numero)
17         } catch (AccesRefuse e) {
18             // On n'avait pas le droit d'utiliser le téléphone
19             // (vérification d'accès a échoué)
20             Logger.getLogger().log(LOG_ERROR, "Acces refuse");
21             throws new RedirectToLoginPage();
22         }
23
24      transaction.commit();
25     } catch (Exception e) {
26         Logger.getLogger().log(LOG_ERROR, "Impossible de composer");
27         if(!db_is_locked) {
28             return;
29         }
30         Logger.getLogger().log(LOG_DEBUG, "Liberation de la table");
31         dao.unlockTable("phone_table");
32
33         if(transaction !=null) {
34             Logger.getLogger().log(LOG_DEBUG, "Renversement de la BD OK");
35             transaction.rollback();
36         }
37
38     }
```

Exemple

```
1  /**
2    *Composer le numéro de téléphone demandé
3    */
4  public void composerTelephone(String numero) {
5      Telephone phone =Telephone.getinstance
```

Préoccupation transverse: Journalisation

```
           try {
16             telephone.composer(numero)
17         } catch (AccesRefuse e) {
18             // On n'avait pas le droit d'utiliser le téléphone
19             // (vérification d'accès a échoué)
20             Logger.getLogger().log(LOG_ERROR, "Acces refuse");
21             throws new RedirectToLoginPage();
22         }
23

26         Logger.getLogger().log(LOG_ERROR, "Impossible de composer");
27

30         Logger.getLogger().log(LOG_DEBUG, "Liberation de la table"

32
33
34             Logger.getLogger().log(LOG_DEBUG, "Renversement de la BD OK");
```

Exemple

```
1  /**
2    *Composer le numéro de téléphone demandé
3    */
4  public void composerTelephone(String numero) {
5      Telephone phone =Telephone.getinstance
```

Préoccupation transverse:
Sécurité

```
           try {
16             telephone.composer(numero)
17         } catch (AccesRefuse e) {
18             // On n'avait pas le droit d'utiliser le téléphone
19             // (vérification d'accès a échoué)

21             throws new RedirectToLoginPage();
22         }
23
```

Exemple

```
1  /**
2     *Composer le numéro de téléphone demandé
3     */
4  public void composerTelephone(String numero) {
5        Telephone phone =Telephone.getinstance
```

Préoccupation PRINCIPALE:
Téléphoner !

```
16                  telephone.composer(numero)
```

Exemple

```
1  /**
2    *Composer le numéro de téléphone demandé
3    */
4  public void composerTelephone(String numero) {
5      Telephone phone =Telephone.getinstance();
6
7      DAO dao =DAO.getCurrentDAO();
8      boolean db_is_locked =false;
9      DBTransaction transaction =null;
10     try {
11         dao.doRequest(new Request.SelectForUpdate("phone_table"));
12         db_is_locked =true;
13         transaction =dao.beginTransaction();
14
15         try {
16             telephone.composer(numero)
17         } catch (AccesRefuse e) {
18             // On n'avait pas le droit d'utiliser le téléphone
19             // (vérification d'accès a échoué)
20             Logger.getLogger().log(LOG_ERROR, "Acces refuse");
21             throws new RedirectToLoginPage();
22         }
23
24      transaction.commit();
25     } catch (Exception e) {
26         Logger.getLogger().log(LOG_ERROR, "Impossible de composer");
27         if(!db_is_locked) {
28             return;
29         }
30         Logger.getLogger().log(LOG_DEBUG, "Liberation de la table");
31         dao.unlockTable("phone_table");
32
33         if(transaction !=null) {
34             Logger.getLogger().log(LOG_DEBUG, "Renversement de la BD OK");
35             transaction.rollback();
36         }
37         return;
38     }
```

Préoccupation PRINCIPALE:
Téléphoner !

Conséquences

- Code dupliqué
- Code plus difficile à comprendre (**clarté**)
- Réutilisation difficile
- Code plus difficile à faire évoluer et à réusiner
  - Ne pas oublier de modifier le code partout
  - Changer de façon de faire →Changer plusieurs lignes dispersées
  - Un changement dans un module peut avoir des répercussions inattendues
- Tests éparpillés et testant autre chose que le but visé par la méthode
- Design plus difficile à élaborer
  - Il faut toujours considérer les effets des préoccupations secondaires

Exemple de risques

- Fort risque d'oublier une vérification d'accès
  - Il faut appeler la vérification dans toutes les méthodes
  - Si on oublie une méthode, il y a une anomalie ou pire... une faille !
- Modifier la façon de vérifier l'accès (signature de la méthode) demande de modifier toutes les méthodes.

Plan

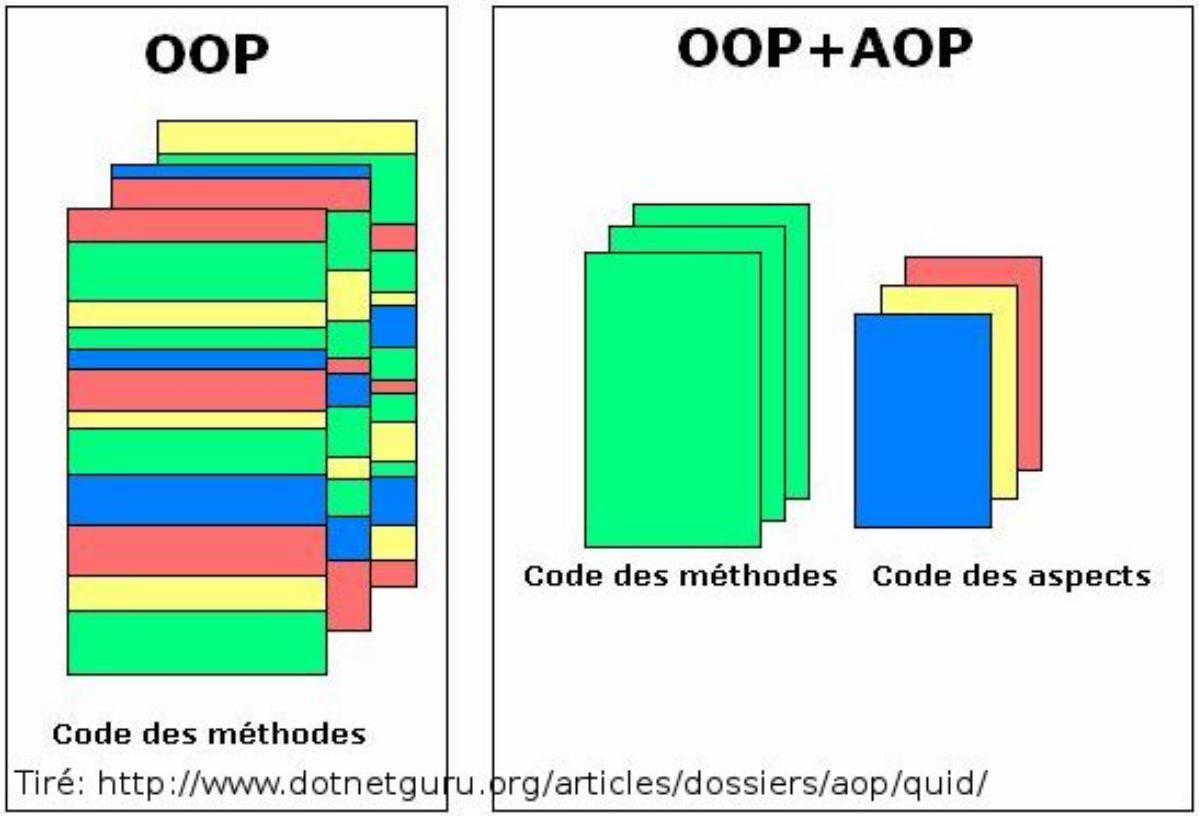
Le paradigme objet (POO)

- Considère le programme comme un groupe d'objets qui collaborent
- Théoriquement, chaque classe devrait représenter une préoccupation
- Aucun mécanisme pour gérer les préoccupations qui s'étendent (préoccupations transverses)

Dans un certain sens, les préoccupations transverses contreviennent à l'objectif initial de la POO qui vise à regrouper dans la même classe une même préoccupation.

Le paradigme aspect

Programmation orientée aspect (AOP)



Le paradigme aspect

Programmation orientée aspect (AOP)

- Programmation Orientée Aspect (POA)
  Aspect Oriented Programming (AOP)
- Inventé par GREGOR KICZALES *et al.* de Xerox PARC
- Conçu afin de répondre au problème des préoccupations transverses en POO
- Permet de regrouper les préoccupations transverses dans des modules indépendants (aspects)
- Les classes du système n'ont pas besoin de connaître les autres préoccupations
- L'aspect décide lui-même où s'exécuter
- Les aspects se lient aux autres modules via un mécanisme de points de jonction.
- Support pour plusieurs langages : Java, .NET, Ruby, Python, etc.

Point de jonction

Join point

- Parfois appelé point de jointure
- Point précis dans le programme
- Emplacements possibles et valide pour injecter du code
- Généralement, des emplacements dans le flot de contrôle

Exemples

- L'appel d'une méthode
- L'accès (get/set) à un attribut
- Lorsqu'une exception est lancée
- Etc

Point de coupure

Point cut

- Identifie les points de jonction où du code (advcies) sera injecté
- Sous-ensemble des points de jonction
- Détermine où l'aspect doit s'exécuter en fonction de certaines conditions parmi tous les points de jonction

Exemple en AspectJ

Tous les appels à une méthode dont le nom débute par « do » sitúés dans une méthode de la classe « TheClass » :

```
1   pointcut callToDoMethodsInTheClass :
2       call(* do*(..)) &&
3       within(TheClass);
```

Conseil (ou greffon)

Advice

- Code à exécuter
- Peut généralement s'exécuter avant, après ou autour d'un point de coupure (AspectJ)

Exemple en AspectJ

Saluer avant les emplacements désignés par le point de coupure « saluerPointcut »

```
1   before() : saluerPointcut() {
2       System.out.prinln("Bonjour le monde!");
3   }
```

Les injections

Inter-type Declarations

- Permettent de modifier une classe sans que celle-ci ne soit au courant
- Exemples :
  - Ajouter des méthodes à une classe
  - Modifier l'héritage

Exemple en AspectJ

Injection de la méthode « acceptVisitor(Visitor v) » à la classe « TheClass »

```
1   void TheClass.acceptVisitor(Visitor v){
2       v.visit(object);
3   }
```

L'aspect

- Un module indépendant (très similaire à une classe dans AspectJ)
- Des règles (points de coupure – pointcuts) qui définissent quand et où l'aspect doit s'activer
- Du code (conseils – advices) qui peut s'éxécuter :
  1. Avant le point de jonction
  2. Après le point de jonction
  3. Autour/en remplacement du point de jonction
- Des injections (inter-type declaration)

Note : Ceci est une vision technique telle qu'utilisée avec AspectJ.
Note : Peut varier en fonction du langage et du tisseur.

Avantages

- Meilleure modularité : Les préoccupations transverses sont isolées dans des modules (aspects)
- Compréhension du code : Pas de pollution par les préoccupations transverses
- Diminution du couplage
- Augmentation de la cohésion
- Facilite le travail collectif

Désavantages

- Technologie jeune (immature)
- Perte de performance
- Peu d'outils supportant le développement
- Méconnaissance concernant l'intégration dans le processus logiciel
- Manque de « bonnes pratiques » reconnues pour l'AOP
- Les aspects peuvent introduire des erreurs
  - Les aspects peuvent être difficiles à tester
  - Difficulté de savoir quels aspects s'exécutent et où
  - Possibilité de conflits entre les aspects
- Complexité (trop c'est comme pas assez !)
- Méconnaissance des répercutions de l'AOP sur l'ensemble d'un projet (méthodologie, tests, réusinage, etc)

Tissage et AspectJ

- Tisseurs d'aspects (Weaver)
  - Tissage (weaving) : Processus d'intégration des aspects et des objets
  - Bref, une sorte de « Compilateur » qui permet d'injecter le code
- Principalement 2 catégories :
  - À la compilation (compile-time et post-compile)
  - Dynamiquement à l'exécution (run-time et load-time)
- En théorie, un langage AOP peut être tissé par plusieurs tisseurs

Tissage et AspectJ (suite)

- AspectJ (langage et tisseur)
  - Extension au langage Java
  - Très évolué
  - Support pour Eclipse (AJDT)
  - Plusieurs types de tissage supportés
- Autres langages/tisseurs :
  - SpringAOP (XML, ...)
  - JBossAOP (Pure-Java)
  - Aquarium (Ruby)
  - AspectC++
  - AspectSharp (Aspect#)
  - Etc.

Plan

Plan

Adoption

- L'industrie utilise de plus en plus l'AOP
  - Souvent utilisé comme outil pour contourner un problème
  - Utilisation encore souvent sporadique
  - Encore peu utilisé comme paradigme (sens large)
- Projets OpenSource (Java) :
  - Terracotta
  - Spring
  - GlassBox
  - Contract4J
  - Etc.

Recherches

- Recherches actives en R&D et dans les universités :
  - Gregor Kiczales, University of British Columbia, Canada
  - IBM Research Lab et Eclipse Fondation (AspectJ)
  - Plusieurs autres universités et centres de recherche
- Comment tester les aspects et le code conseillé
- Méthodologies de développement et intégration dans le processus de conception du logiciel
  - Aspect Oriented Software Developpement (AOSD)
- Élaboration de « Patterns » et de bonnes pratiques
- Réusinage (refactoring) de code AOP
- Développement d'outils pour supporter le dev. AOP (IBM Research, AspectJ, ...)

Plan

Injection dans un « framework »

Exemple : ArrayList

Exemple

Nous utilisons une *ArrayList* en Java

1. On veut écrire dans la console un message à chaque fois qu'un élément est ajouté à la liste.
2. On ne veut jamais pouvoir ajouter le chiffre 0 à la liste.

```
1  /* --- Main --- */
2  List<Integer> maListe = new ArrayList<Integer>();
3  maListe.add(1);
4  maListe.add(0);
5  maListe.add(6);
6
7  System.out.println("Ma liste = " + maListe.toString());
```

Note : Dans ce cas, il serait préférable d'utiliser l'héritage.

Injection dans un « framework »

Autres exemples

- Forcer un « framework » web à retenir la réponse en attendant une confirmation
- Extraire des informations dans une bibliothèque
- Faire un « Adapter » entre deux bibliothèques sans les modifier
- Modifier ou ajouter des comportements quand la biliothèque ne permet pas l'héritage (mauvaise gestion du polymorphisme ou non-respect du Open-Closed Principle)
- Réagir à des événements qui n'ont pas été prévus pour être observables (pas de gestion des plugiciels ou support incomplet)
  - Exemple : Écrire un plugiciel Eclipse qui allume des « LEDs » vertes ou rouges en fonction du résultat des tests unitaires. Mais le plugiciel « Eclipse Junit » n'est pas précis entre un test avec le statut « ERROR » vs « FAILURE ».

Pollution des signatures

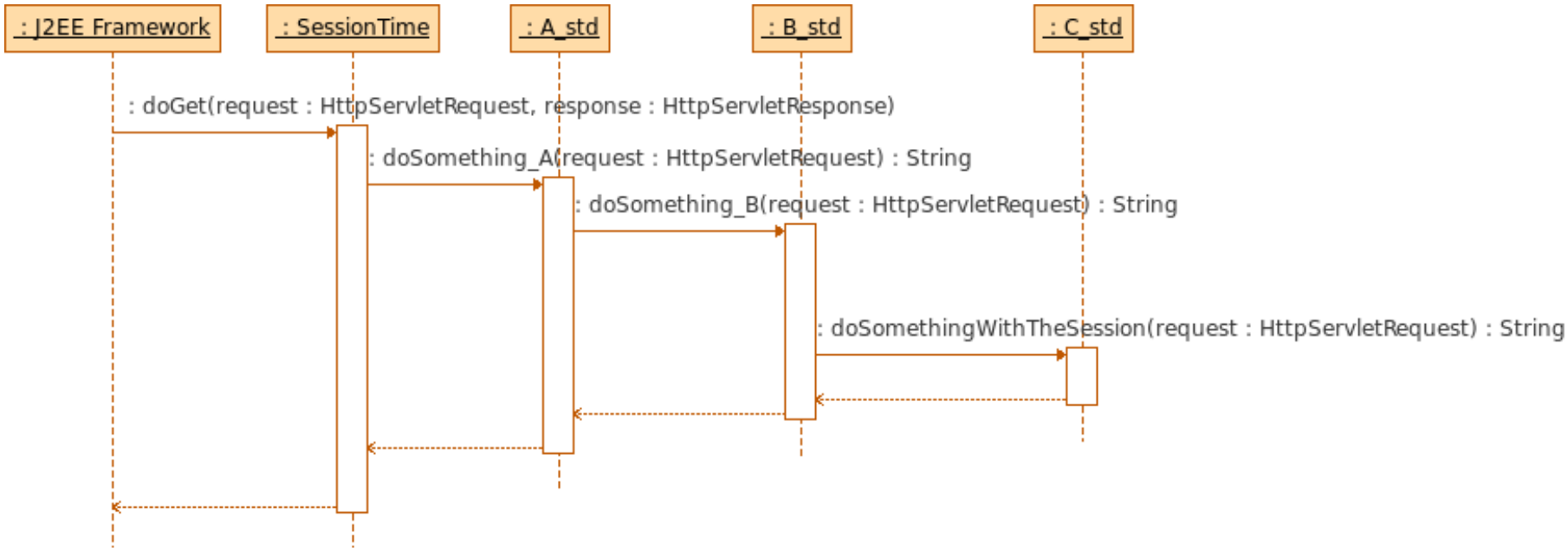
Mise en contexte

- Éviter de passer des paramètres de classes en classes alors que très peu utilisent réellement le paramètre.

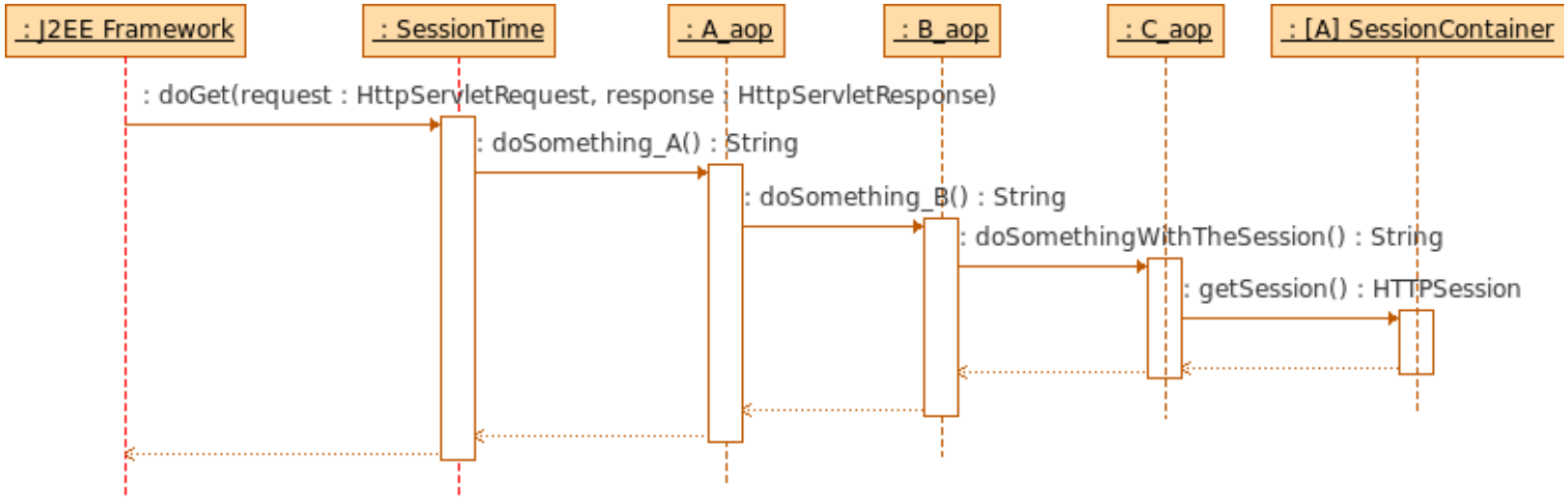
Exemple

Objet représentant une session dans une application web.

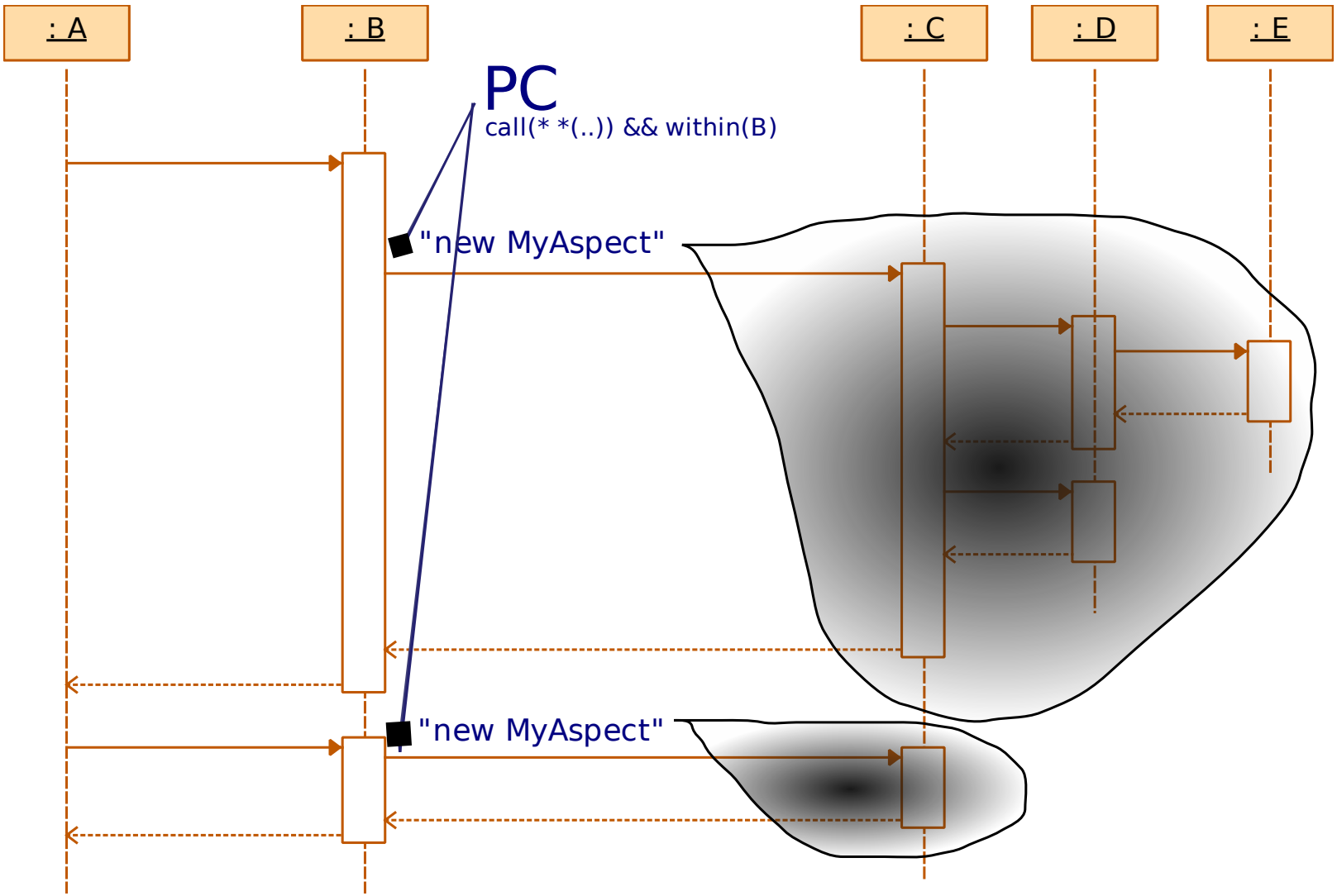
Sans AOP :



Avec AOP :



Pollution des signatures (suite)



Gestion de la concurrence

Exemple 1

Exemple 1 : Nettoyage prioritaire

- Le système est composé de plusieurs commandes qui peuvent exécuter une action.
- Une classe « Cleaner » est chargée de faire le nettoyage de la B.D.
- 2 « threads » :
  1. Runner : Exécute des commandes
  2. Cleaner : Fait le ménage à toutes les 5 secondes
- Règles :
  1. Tout peut être exécuté en parallèle sauf lors du nettoyage
  2. Lors du nettoyage, aucune commande ne peut être exécutée
  3. Sauf si la commande est issue du nettoyage (dans le flot d'appels de Cleaner.clean)

Gestion de la concurrence

Exemple 1

```
1  public class CopyCommand implements Command {
2      @Override
3      public void execute() {
4          System.out.println("[" + Thread.currentThread().getName() + "] EXECUTE Copy");
5      }
6  }
```

```
1  public class Cleaner {
2      // [...]
3      public void clean() {
4          System.out.println("[" + Thread.currentThread().getName() + "] -------- BEGIN CLEAN ---------");
5
6          redo.execute();
7
8          try {
9              Thread.sleep(5000);
10         } catch (InterruptedException e) {
11         }
12
13         System.out.println("[" + Thread.currentThread().getName() + "] -------- END CLEAN ---------");
14     }
15 //[...]
```

Gestion de la concurrence (suite)

Exemple 1

```
1  public aspect LockOnCleaning {
2      Boolean isLocked = false;
3
4      protected pointcut cleaning() :
5          execution( void Cleaner.clean() );
6
7      protected pointcut executeCommand() :
8          execution( void Command+.execute(..) );
9
10     before() : cleaning() {
11         synchronized (isLocked) {
12             isLocked = true;
13     //}}
14
15     after() : cleaning() {
16         synchronized (isLocked) {
17             isLocked = false;
18     //}}
19
20     before() : executeCommand() && !cflow(cleaning()) {
21         synchronized (isLocked) {
22             while (isLocked) {
23                 isLocked.wait(500); // [TRY...CATCH(InterruptedException )]
24             }
25 //}}}
```

Gestion de la concurrence (suite)

Exemple 1

```
1  ============= SANS AOP ===============
2
3  [Runner] EXECUTE Copy
4  [Runner] EXECUTE Redo
5  [Cleaner] ---------- BEGIN CLEAN ----------
6  [Cleaner] EXECUTE Redo
7  [Runner] EXECUTE Copy
8  ...
9  [Runner] EXECUTE Copy
10 [Runner] EXECUTE Redo
11 [Cleaner] ---------- END CLEAN ----------
12
13 ============= AVEC AOP ===============
14
15 [Runner] EXECUTE Copy
16 [Runner] EXECUTE Redo
17 [Cleaner] ---------- BEGIN CLEAN ----------
18 [Cleaner] EXECUTE Redo
19 [Cleaner] ---------- END CLEAN ----------
20 [Runner] EXECUTE Copy
21 [Runner] EXECUTE Redo
```

Gestion de la concurrence

Exemple 2

Exemple 2 : Variation sur les lecteurs et les écrivains

- Une classe fichier a des attributs
- L'accès aux attributs est indépendant (la lecture/écriture d'un attribut n'a pas d'effet sur les autres)
- Règles :
  1. On peut lire en même temps
  2. Un seul écrivain peut écrire à la fois (pour un attribut)
  3. On ne peut pas lire pendant qu'une écriture est en cours

Gestion de la concurrence

Exemple 2

```
1  public class Fichier {
2
3      private volatile int foo;
4      private volatile int bar;
5
6      public int getFoo() { return foo; }
7
8      public void setFoo(int foo) {
9          this.foo = foo;
10         try {
11             Thread.sleep(2000); // Longue opération...
12         } catch (InterruptedException e) { }
13     }
14
15     public int getBar() { return bar; }
16
17     public void setBar(int bar) {
18         this.bar = bar;
19         try {
20             Thread.sleep(2000); // Longue opération...
21         } catch (InterruptedException e) { }
22     }
23 }
```

Gestion de la concurrence (suite)

Exemple 2

```
1  @Singleton //<<< Singleton version AOP
2  public class LockManager {
3
4      Map<String, Boolean> locks = new HashMap<String, Boolean>();
5
6      public synchronized boolean isLock(String key) {
7          Boolean isLocked = locks.get(key);
8          if (isLocked == null)
9              isLocked = false;
10         return isLocked;
11     }
12
13     public synchronized boolean acquireLock(String key) {
14         if (isLock(key))
15             return false;
16         locks.put(key, true);
17         return true;
18     }
19
20     public synchronized void releaseLock(String key) {
21         locks.put(key, false);
22     }
23 }
```

Gestion de la concurrence (suite)

Exemple 2

```
1  public aspect ConcurrenceFichier {
2
3      protected pointcut writeAttribute() :
4          execution( void Fichier.set*(..) );
5
6      protected pointcut readAttribute() :
7          execution( * Fichier.get*(..) );
8
9      Object around() : writeAttribute() {
10         String threadName = Thread.currentThread().getName();
11         String methodName = thisJoinPoint.getSignature().getName();
12         String typeToBeWritten = methodName.substring(3);
13         LockManager lockManager = new LockManager();
14
15         boolean hasLock = lockManager.acquireLock(typeToBeWritten);
16         while (!hasLock) {
17             System.out.println("[" + threadName + "] WAITING to WRITE "
18                     + typeToBeWritten);
19             try {
20                 Thread.sleep(500);
21             } catch (InterruptedException e) {
22                 continue;
23             }
24
25             hasLock = lockManager.acquireLock(typeToBeWritten);
26         }
```

Gestion de la concurrence (suite)

Exemple 2

```
27
28          Object ret = proceed();
29
30          if (hasLock)
31              lockManager.releaseLock(typeToBeWritten);
32
33          return ret;
34      }
35
36      Object around() : readAttribute() {
37          // [...]
38      }
39
40  }
```

Gestion de la concurrence

Autres exemples

- Terracotta
- Portefeuille financier :
  - Lecture synchronisée des données
  - « Thread » indépendant de mise à jour des données boursières

Autres utilisations possibles

- Détection d'erreurs (Partie 2)
- Forcer le respect d'une norme de programmation/convention (Partie 2)
- Redondance et réseautique
  - Serveurs de relève (« failover »)
  - Serveurs de répartition (« load-balancing »)
  - Changer le flot de contrôle si la connexion réseau est perdue
  - Se replier sur une « cache » si l'information n'est pas disponible
  - Changer de sources de données (serveur) dynamiquement
  - Vérifications redondantes (2 serveurs doivent donner le même résultat)

Autres utilisations possibles (suite)

- Architecture
  - Design patterns version AOP (Partie 2)
- Vérifications et tests
  - Injection d'assertions
  - Prévention de « deadlock »
  - Injection de « Mocks » (Partie 2)
  - Tests unitaires utilisant de l'AOP (Partie 2)
  - Profilage : démarrer arrêter le chronomètre mais uniquement en mode profilage et sans polluer le code. (exemple : JRAT)
- Etc.

Deuxième partie

Deuxième partie

Impacts de l'AOP sur l'architecture et la qualité du logiciel

Une brève introduction