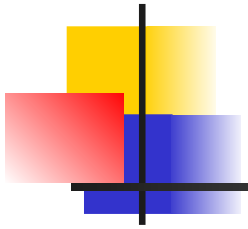
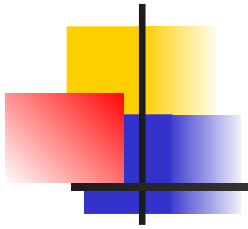


CHAPITRE VI:
PARADIGME ORIENTÉ ASPECT (POA)

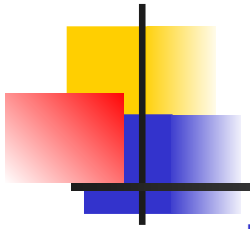
Sommaire:

1. Introduction
2. Développement logiciel orienté aspect
3. Concepts de la POA
4. Conclusion



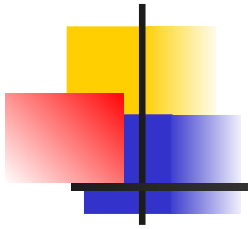
1. Introduction

- Les bénéfices de la POO pour le développement de logiciels complexes sont indéniables (diverses expériences).
- Cependant, dans certains cas, la POO ne fournit pas de solutions satisfaisantes : fonctionnalités transversales et dispersion du code.
- Depuis 1997, le paradigme orienté aspect provoque l'engouement de la communauté scientifique s'intéressant au GL (différents milieux).
- Cette nouvelle approche offre de nouvelles perspectives à un principe bien connu en GL : la séparation des préoccupations transverses : Avenue prometteuse.



Fonctionnalités transversales

- L'approche OO a pour objectif de découper une application en classes : séparation des données et des traitements associés dans des entités cohérentes.
- Bien qu'indépendantes, ces entités peuvent parfois être corrélées – contraintes d'intégrité référentielle.
- **Exemple :** objet client ne doit pas être supprimé tant que sa commande n'a pas été honorée.
  - Solution 1 : consiste à modifier la méthode *supprimer* de la classe client afin de vérifier au préalable l'absence de la commande non honorée – mauvaise solution.
  - Solution 2 : permettre à la classe *commande* de le faire – mauvaise solution.
- La contrainte, dans ce cas, est transversale à ces deux entités.
- Les fonctionnalités transversales (contraintes d'intégrité) brisent l'indépendance des classes.
- La POO ne fournit aucune solution pour ce type de problème.
- Le développement d'applications complexes : démonstration des limites éprouvées par la POO.



Dispersion du code

En POO, la modification portant sur le code de la méthode est transparente pour tous les objets l'utilisant.

- La modification de la signature de la méthode ne l'est pas – il est nécessaire de modifier toutes les classes l'utilisant.
  - Modification coûteuse si la méthode est largement utilisée.
- En POO, l'implémentation d'une méthode est localisée dans une classe.
- Son invocation, par contre, est dispersée dans plusieurs classes.
  - La dispersion du code réduit la maintenance et l'évolution des applications orientées objet.

Fonctionnalités Transversales et Dispersion du code

- La modélisation logicielle et le développement sont affectés à différents niveaux :

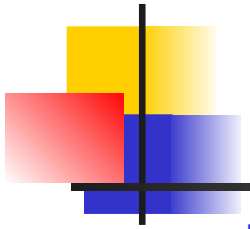
Mauvaise traçabilité : implémentation simultanée de multiples contraintes – correspondance non claire entre contrainte et implémentation.

Faible productivité : éloignement du développeur du but final en raison de l'implémentation simultanée des contraintes multiples.

Réutilisation du code faible : module avec plusieurs responsabilités.

Code de qualité pauvre : code embrouillé – cohésion et modularité faibles (entre autres).

Évolution difficile : vue et ressource limitées pour répondre aux problèmes immédiats – répondre aux problèmes futurs nécessitent un profond remaniement.

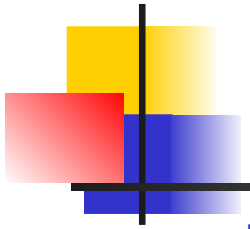


(suite)

- Par ailleurs, selon les principes de la conception :
  - - *Abstraction*
    - - Focus sur les propriétés importantes du système.
  - **-** *Décomposition*
    - - Diviser l'application en modules séparés et adressables.
  - **-** *Encapsulation et information cachée*
    - - Regroupement des propriétés communes.
    - - Cacher les détails d'implémentation.
  - **-** *Séparation des préoccupations*
    - - Faible Couplage
      - Minimiser le couplage entre les composants
    - - Forte cohésion

Séparation des préoccupations

- **Cohésion**
  - - *Maximiser* la cohésion au sein d'un composant
    - - Composant cohésif réalise *un seul rôle*.
    - - Les changements requis peuvent être facilement localisés et ne sont pas propagés.
- **Couplage**
  - - Des composants fortement couplés possèdent plusieurs dépendances/interactions.
  - - *Minimiser* le couplage entre les composants
    - - Réduction de la complexité des interactions.
    - - Réduction des effets de cascade.
- Avantages de la séparation des préoccupations
  - - Compréhensibilité
  - - Maintenabilité
  - - Extensibilité
  - - Réutilisabilité
  - - Adaptabilité



Séparation des préoccupations (suite)

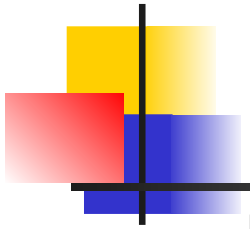
- La séparation des préoccupations au sein d'une application à un impact direct sur les attributs de qualité.

- Solution : Une réponse possible, la POA

- La programmation Orientée Aspect (POA) introduit un nouveau concept, l'aspect – définie en 96 par Grégor Kiczales et son équipe de recherche.

- Exemple de préoccupations transverses :

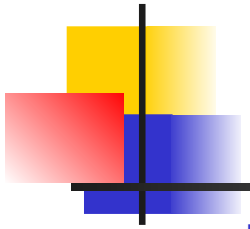
  - Synchronisation
  - Contraintes en temps réel
  - Contraintes d'interaction (design pattern, etc.)
  - Gestion de mémoire
  - Persistance
  - Sécurité
  - Logging
  - Procédure de tests
  - Optimisation spécifique



2. Développement Logiciel Orienté Aspect

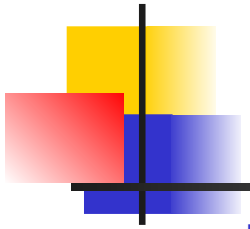
Nomenclature aspect (littérature)

- Crosscutting concern (préoccupation transverse)
  - Une préoccupation qui implique ou touche plusieurs composants
  - Duplication de code épars et entremêlé
  - Expression des besoins non-fonctionnels (BD, gestion du GUI)
- Scattering concern (épars)
  - Une préoccupation qui affecte plusieurs composantes
- Tangling concern (entremêlé)
  - Plusieurs préoccupations sont entremêlées dans un même module (attribution de plusieurs rôles)



Avantages de la POA

- Permet une meilleure séparation des préoccupations transverses.
- Cette amélioration est réalisée à l'aide d'abstractions permettant de **représenter de manière explicite** une préoccupation transverse, i.e. **aspects**.
- Amélioration de la modularité des programmes.
- Augmentation de la productivité.
- Réduction de la complexité du projet.
- Et amélioration de certains critères de qualité tels que :
  - maintenabilité, traçabilité, réutilisabilité, compréhensibilité...



Composition d'une application OA

- Une application OA (combinaison objet-aspect) est composée de deux parties :

  1. Les classes : regroupent les données et les traitements pour répondre aux besoins de l'application

  2. Les aspects intègrent aux classes des éléments supplémentaires. Ces éléments correspondent à des fonctionnalités transversales ou à des fonctionnalités dont l'utilisation est dispersée

- La principale force des aspects réside dans leur capacité à étendre le comportement des objets

- Les exigences non fonctionnelles (parfois fonctionnelles) se retrouvent souvent dans diverses classes. Les aspects permettent de les regrouper dans des unités modulaires appelées aspects.

Composition d'une application OA

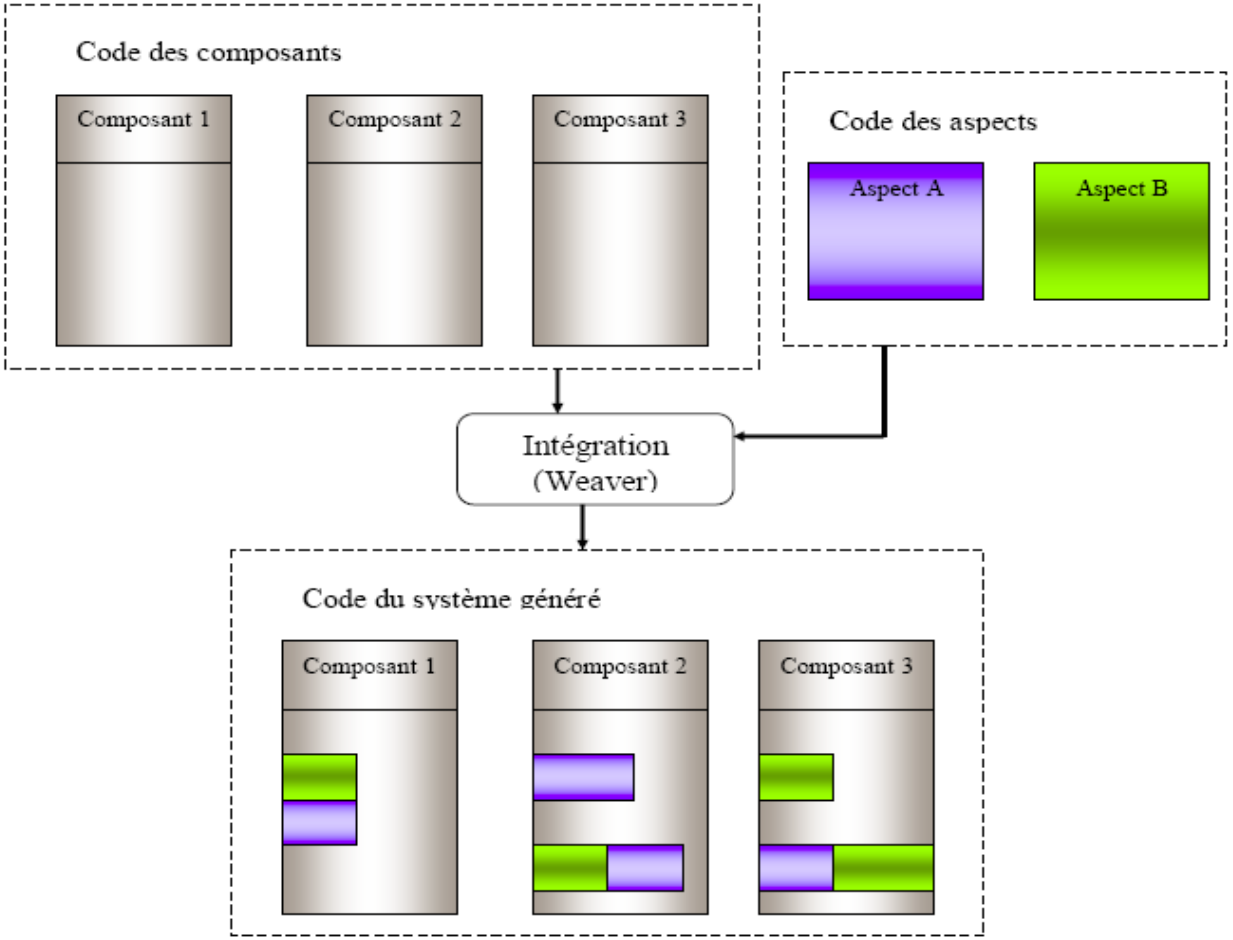


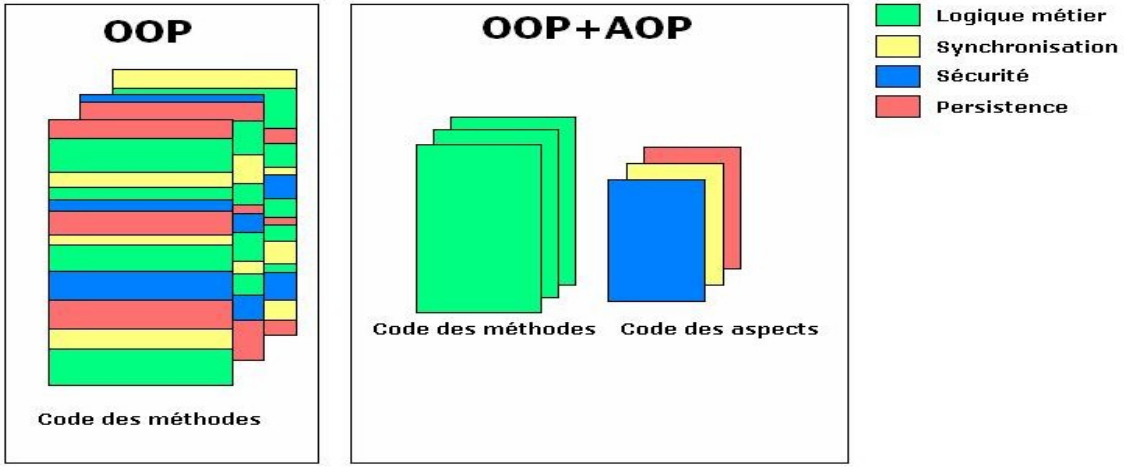
Figure 1 : Extraite de Baltus J. 02

Composition d'une application OA (suite)

- L'opération permettant l'intégration des fonctionnalités des classes et celle des aspects pour obtenir une application opérationnelle peut se faire soit de façon statique (AspectJ) soit de façon dynamique (JAC – Java Aspect Components)

- Le développement en POA se résume en trois étapes :

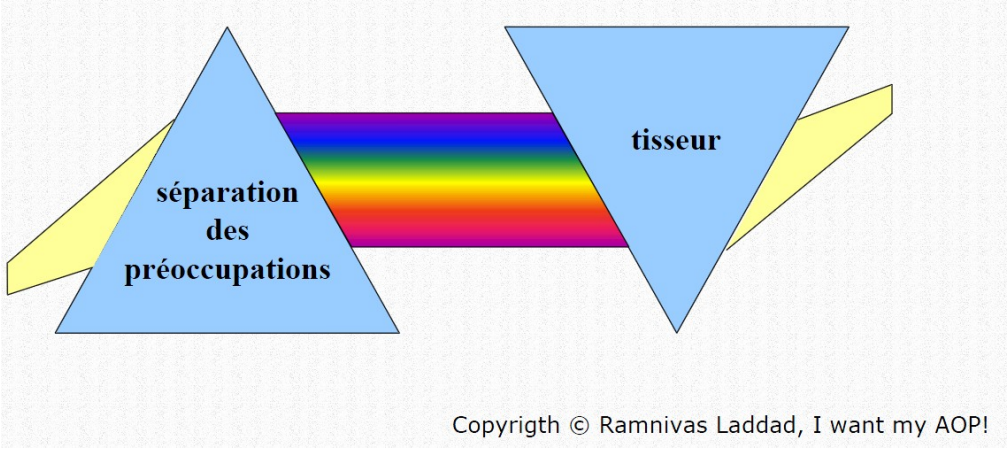
1. Décomposition des éléments ou séparation des préoccupations



Composition d'une application OA (suite)

2. Implémentation de chaque préoccupation – aspect

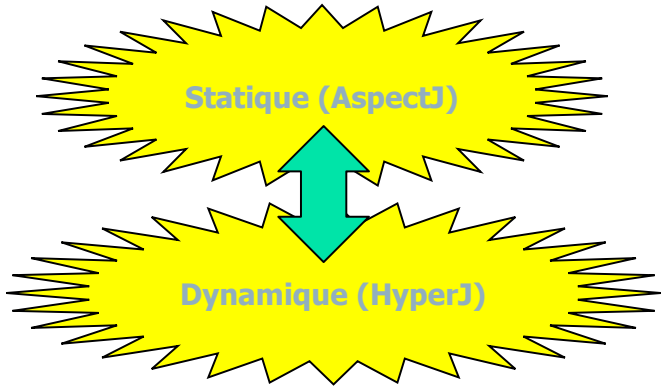
3. Intégration du système



Compilation des aspects (weaving)

- Les diverses implémentations AOP utilisent des approches très diversifiées

- Avant la compilation (pré-processeur)
- Durant la compilation
- Après la compilation
- Durant le chargement de l'application
- Durant l'exécution (Just in Time)



- Les besoins de l'application définissent généralement la « bonne» solution à utiliser

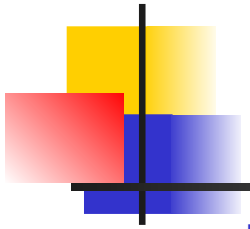
- Technologie AOP disponibles

- AspectWerkz
  http://aspectwerkz.codehaus.org/

- AspectJ
  http://eclipse.org/aspectj/

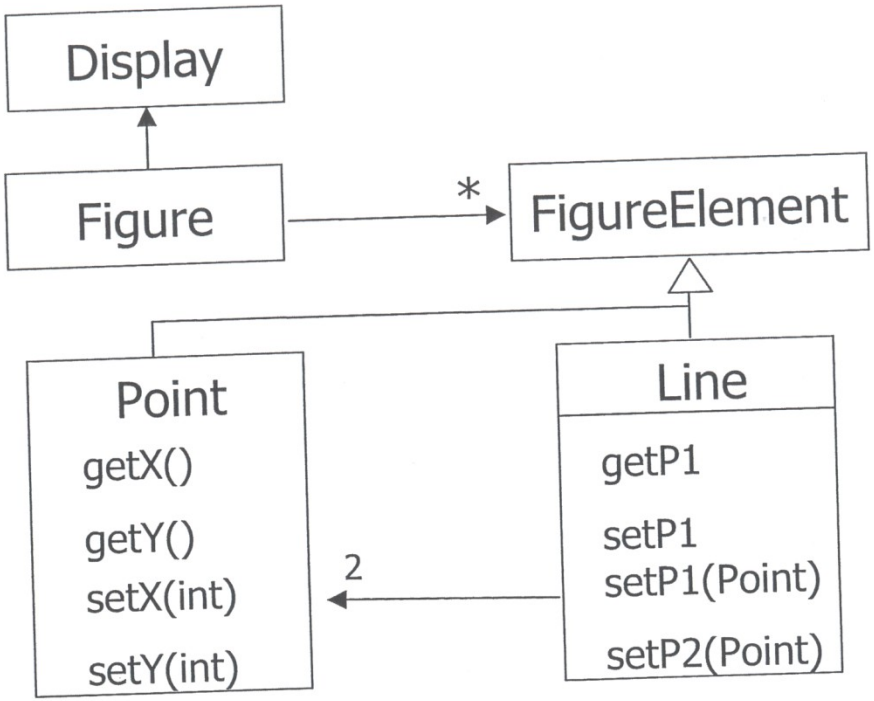
- DemeterJ/DJ
  http://www.ccs.neu.edu/research/demeter/DemeterJava/

- Multi-dimensional separation of Concerns/HyperJ
  http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm



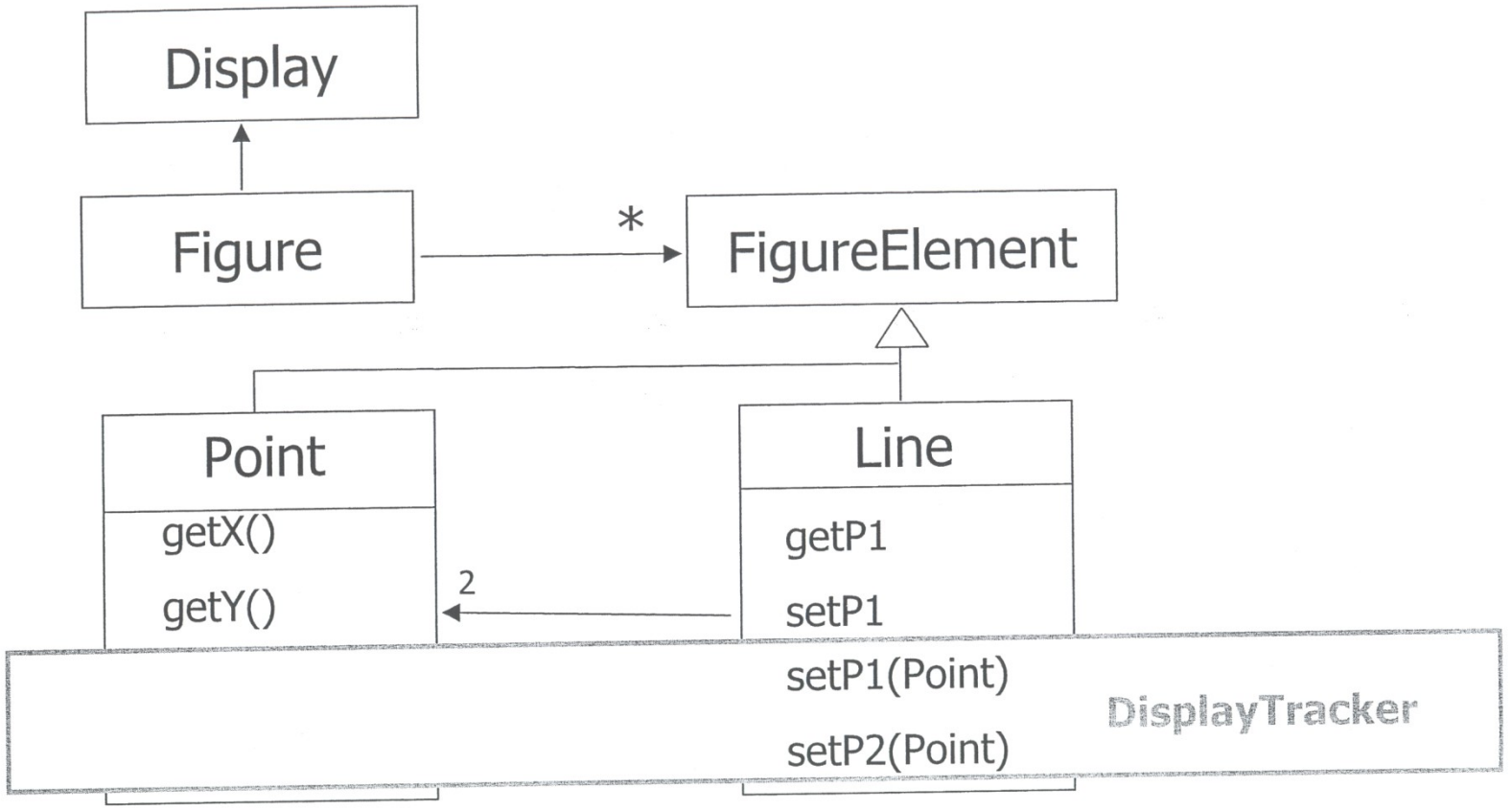
Exemple – Éditeur de figures

- Une *figure* est constituée de plusieurs éléments de *figures*. Un *élément de figure* est soit un *point* ou une *ligne*. Les *Figures* sont tracées sur l'appel de la méthode *Display*. Un *point* est composé de deux coordonnées X et Y. Une *ligne* est définie par deux points.



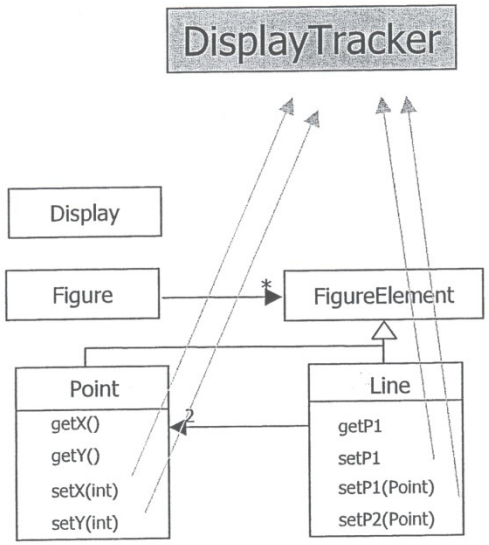
Les Composants sont
- Cohésifs
- Faiblement Couplés
- Possèdent des interfaces définies (abstraction, encapsulation)

Envoyer un message à *ScreenManager* si un élément de figure est déplacé



Résolution OO traditionnelle

Préoccupation



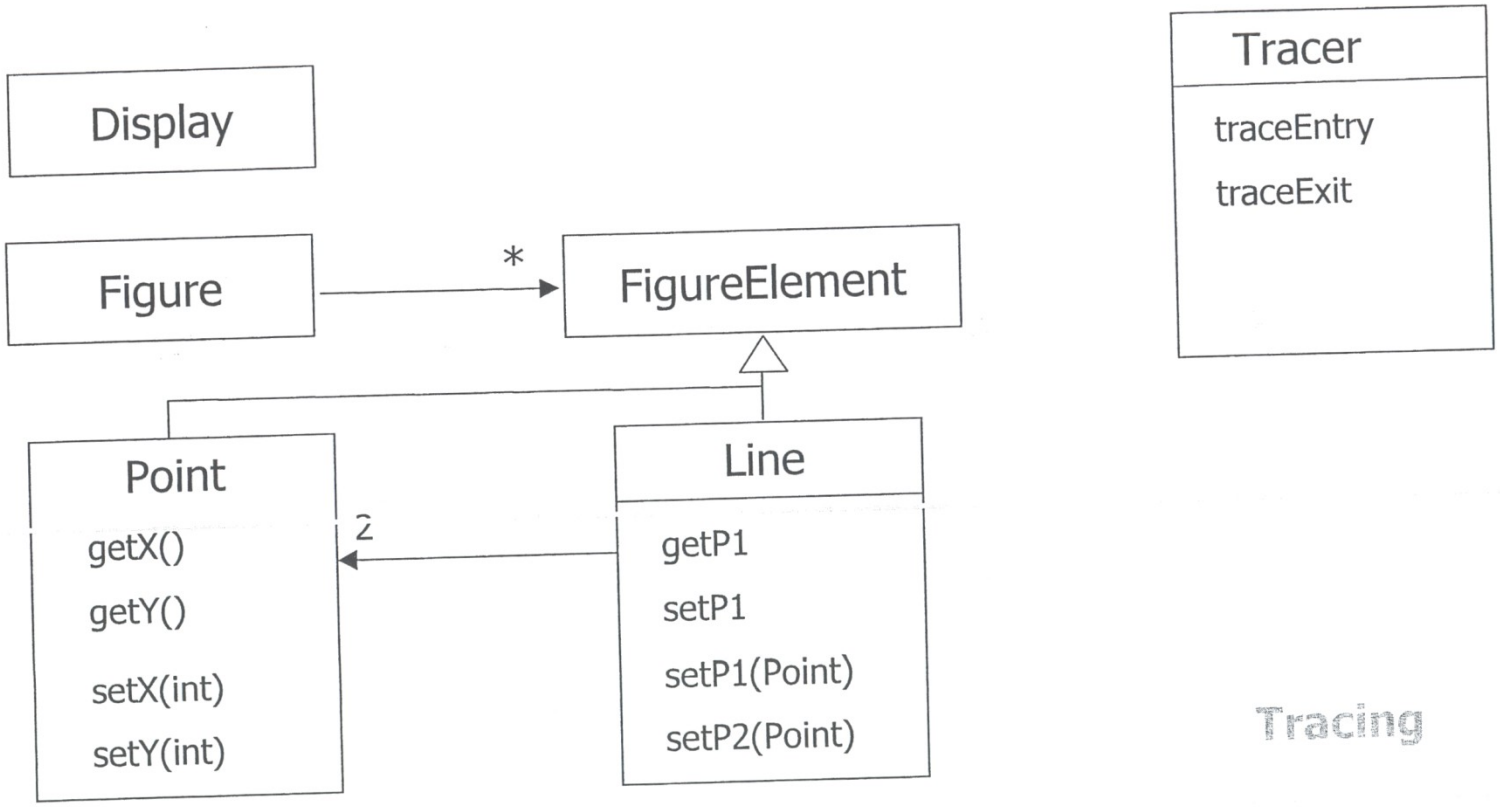

```
class DisplayTracker {

  static void updatePoint(Point p)
  {
        this.display(p);
        ....
  }
  static void updateLine(Line l)
  {
        this.display(l);
        ....
  }
```

```
class Point {
  void setX(int x) {
     DisplayTracker.updatePoint(this);
     this.x = x;

  }
}
```

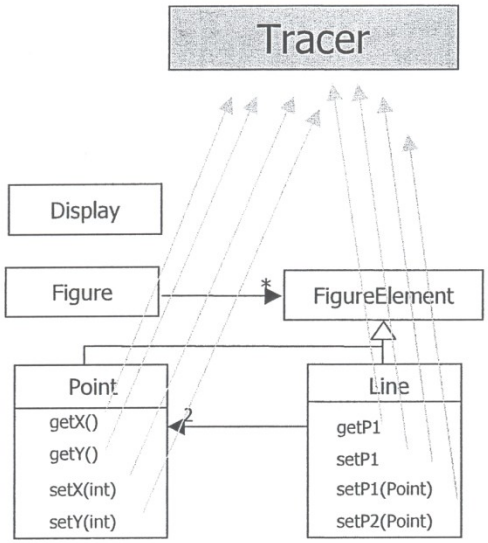
```
class Line {
  void setP1(Point p1 {
     DisplayTracker.updateLine(this);
     this.p1 = p1;

  }
}
```

Exemple d'une 2ième préoccupation

Conserver une trace des opérations exécutées...



Résolution OO tradionnelle




```
class Tracer {

  static void traceEntry(String str)
  {
        System.out.println(str);
  }
  static void traceExit(String str)
  {
        System.out.println(str);
  }
}
```

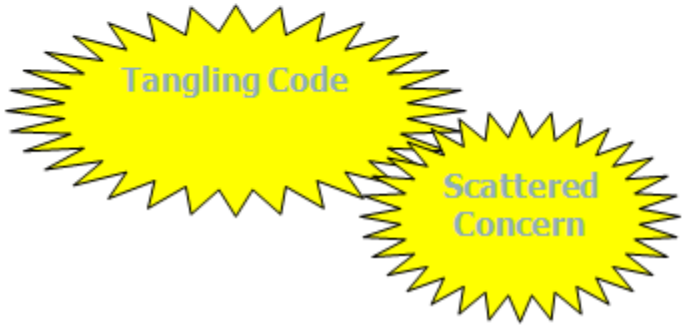
```
class Point {
 void setX(int x) {
 Tracer.traceEntry("Entry Point.set");
   _x = x;
 Tracer.traceExit("Exit Point.set");
}
}
```

```
class Line {
  void setP1(Point p1 {
    Tracer.traceEntry("Entry Line.set");
    _p1 = p1;
    Tracer.traceExit("Exit Line.set");
}
}
```

Rappel – Sans l'AOP

```
class Line {
 private Point _p1, _p2;

 Point getP1() { return _p1; }
 Point getP2() { return _p2; }

 void setP1(Point p1) {
   Tracer.traceEntry("entry setP1");
  _p1 = p1;
  Tracer.traceExit("exit setP1");
 }

 void setP2(Point p2) {
   Tracer.traceEntry("entry setP2");
  _p2 = p2;
  Tracer.traceExit("exit setP2");
 }
class Point {
 private int _x = 0, _y = 0;

 int getX() { return _x; }
 int getY() { return _y; }

 void setX(int x) {
  Tracer.traceEntry("entry setX");

   _x = x;
  Tracer.traceExit("exit setX")
 }
 void setY(int y) {
  Tracer.traceEntry("exit setY");

   _y = y;
  Tracer.traceExit("exit setY");
 }
}
```

```
class Tracer {

  static void traceEntry(String str)
  {
       System.out.println(str);
  }
  static void traceExit(String str)
  {
       System.out.println(str);
  }
}
```



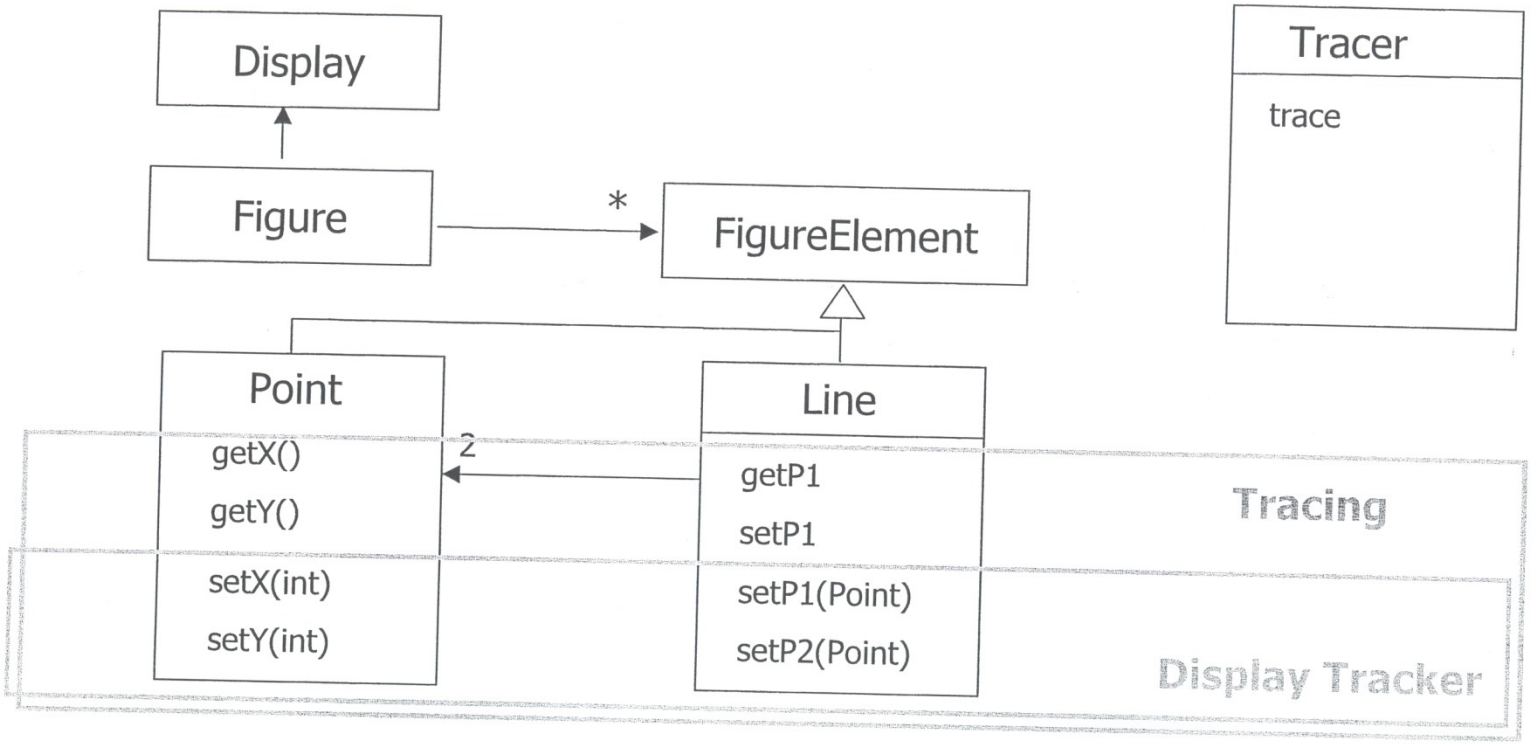
Exemple – Avec AOP

```
class Line {
  private Point _p1, _p2;

  Point getP1() { return _p1; }
  Point getP2() { return _p2; }

  void setP1(Point p1) {
    _p1 = p1;
  }
  void setP2(Point p2) {
    _p2 = p2;
  }
}

class Point {
  private int _x = 0, _y = 0;

  int getX() { return _x; }
  int getY() { return _y; }

  void setX(int x) {
    _x = x;
  }
  void setY(int y) {
    _y = y;
  }
}
```

```
aspect Tracing {

  pointcut traced():
    call(* Line.* ||
    call(* Point.*);

  before(): traced() {
    println("Entering:" +
        thisjopinpoint);

  void println(String str)
  {<write to appropriate stream>}

  }
}
```

- Un Aspect est définit dans un module séparé
- La préoccupation est localisée
- Pas de code dupliqué
- Amélioration de la cohésion des classes

Exemple – Tracing ? Display ?



3. Concepts de la POA

- Notion d'aspect
  - Exemple de dispersion du code : gestion des traces

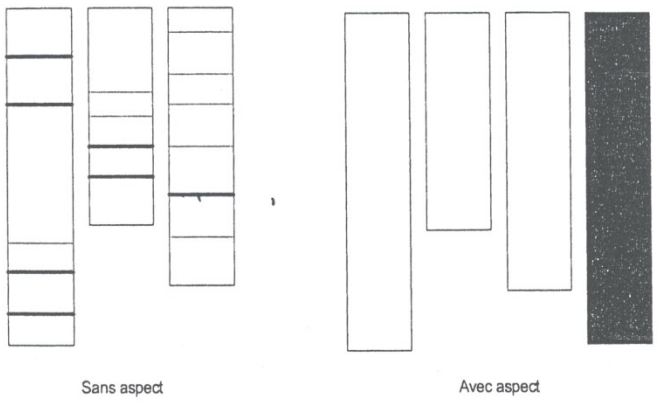


Figure 4 :

- En cas de modification, il est nécessaire de parcourir l'ensemble de ces appels
- La dispersion du code est un frein à la maintenabilité et à l'évolutivité des applications
- La POA fournit un moyen de rassembler dans un aspect du code qui autrement serait dispersé au sein de l'application

3. Concepts de la POA (suite)

- *Définition :* Entité logicielle qui capture une fonctionnalité transversale à une application
  - Unité modulaire représentant une préoccupation traverse.
  - Comme une classe, un aspect peut définir des méthodes, des attributs.
  - Peut être abstrait, hériter de classes et d'aspect abstraits et implémenter des interfaces.
  - Peut introduire de nouvelles méthodes / attributs à un composant.
- En POA, une application comporte des classes et des aspects – deux dimensions de modularité (fonctionnalités implémentées par les classes et préoccupations transversales implémentées par les aspects)
  - Services non fonctionnels et aspects.
- Application = services fonctionnels + services non fonctionnels
  - Services fonctionnels : partie métier – comportement concret de l'application.
  - Services non fonctionnels : services supplémentaires - problématiques techniques - appelés à de nombreux endroits du code métier de l'application.

Inversion des dépendances

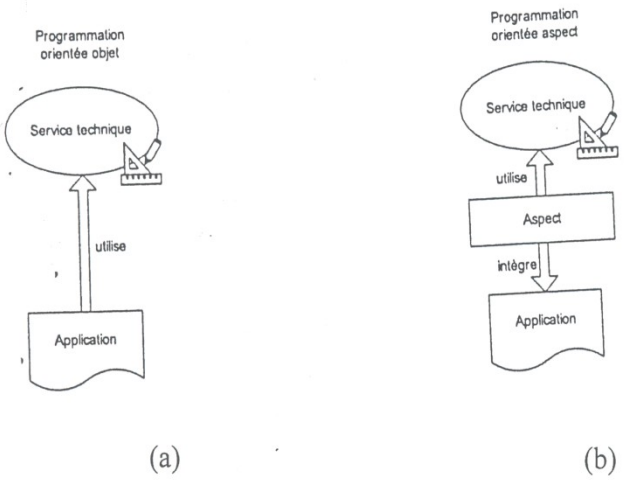
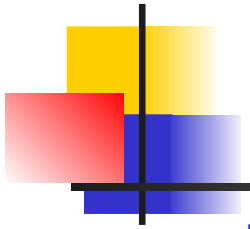


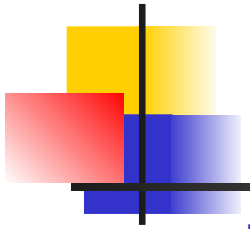
Figure 5 : Inversion des dépendances entre l'application et l'API technique (extrait de programmation orientée aspect pour Java/J2EE , R. Pawlak et al.)

- Figure a : dépendance entre l'application et le service technique utilisé
- Figure b : un aspect utilise un service technique pour l'intégrer à une application – dépendance inversée



Les points de jonction (join point)

- *Définition :*
  - Balises dans le code du programme pour l'exécution de l'aspect
  - Point dans l'exécution d'un programme autour duquel un ou plusieurs aspects peuvent être ajoutés
  - Point de jonction : un appel de méthode, un appel du constructeur, un accès en lecture ou en écriture et les exceptions sont les plus courants et les plus utilisés en POA

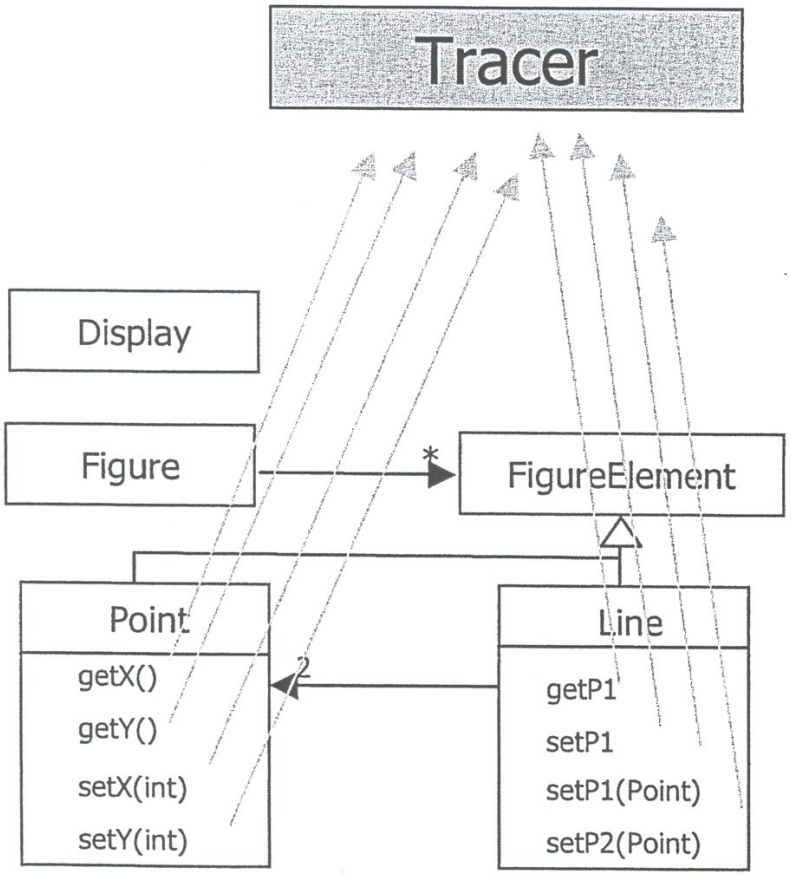


Les coupes (pointcut – crosscut)

- *Définition :* désigne un ensemble de points de jonction. La coupe est définie à l'intérieur d'un aspect

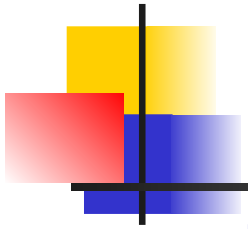
- JoinPoints et Pointcut

  - Joinpoints: **call(* Line.*)** , **call(* Point.*)**
  - Pointcut: collection de joinpoints : **traced()**



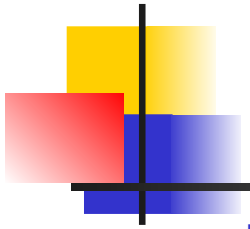

```
Pointcut Traced ():
    call(* Line.*) ||
    call (* Point.*);
```

On veut un point d'intérêt sur tous les appels de méthodes des instances de Line et Point.



Code advice

- Le code advice : bloc de code définissant le comportement d'un aspect – code à exécuter selon les conditions définies dans les coupes
- Un aspect comporte un ou plusieurs code advice – chaque code advice définit un comportement particulier pour son aspect
- Il joue en quelques sortes le même rôle qu'une méthode
- Chaque code advice est associé à une coupe – la coupe fournit l'ensemble des points de jonction autour desquels sera greffé le code advice donc à des points de jonction et possède un type
- Une même coupe peut être utilisée par plusieurs code advice => différents traitements seront greffés autour des mêmes points de jonction => problème de composition d'aspect
- Différents types de code advice



Code advice (suite)

- Le code définit par l'Advice s'exécute :
  - *before*, le code est injecté avant le joinpoint
    before (args): pointcut { Body }
  - - *after*, le code est injecté après le joinpoint
    after (args): pointcut { Body }
  - *around,* le code est injecté around (à la place) du code définit au joinpoint
    ReturnType around (args): pointcut { Body }

Example - AspectJ

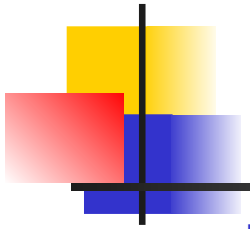
```
class Line {
  private Point _p1, _p2;

  Point getP1() { return _p1; }
  Point getP2() { return _p2; }

  void setP1(Point p1) {
    _p1 = p1;
  }
  void setP2(Point p2) {
    _p2 = p2;
  }
}

class Point {
  private int _x = 0, _y = 0;

  int getX() { return _x; }
  int getY() { return _y; }

  void setX(int x) {
    _x = x;
  }
  void setY(int y) {
    _y = y;
  }
}
```

```
aspect Tracing {

  pointcut traced():
    call(* Line.* ||
    call(* Point.*);

  before(): traced() {
    println("Entering:" +
            thisjopinpoint);

  after(): traced() {
    println("Exit:" +
            thisjopinpoint);


  void println(String str)
  {<write to appropriate stream>}

  }
}
```

aspect

pointcut

advice

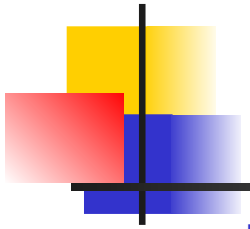


Mécanisme d'introduction

- Le mécanisme d'introduction permet d'étendre le comportement d'une application – ajout d'éléments tels que des attributs ou des méthodes.
- Le mécanisme d'introduction est sans condition – extension réalisée dans tous les cas.
- Exemple : introduction d'un attribut de type date : ajoute un attribut de type date pour enregistrer la date de création de chaque instance.

Composition d'aspects : Interaction entre aspects

- Utilisation de plusieurs aspects; conflits pouvant survenir entre les aspects.
- Les cas de conflits possibles sont :
  - Incompatibilité
  - Dépendance
  - Redondance



Conclusion

- Il y a beaucoup de points à régler …
  - Représentation d'aspects sous UML
  - Identification et intégration des aspects au processus de développement (early aspects)
  - Limites et définition du rôle d'un aspect dans une application.

- L'AOP constitue, à l'heure actuelle, un des secteurs de recherche du génie logiciel les plus actifs.
- Certaines distributions sont utilisées dans les phases de développement mais l'approche reste tout de même expérimentale.
- Plusieurs défis (recherche, expérimentation, etc.).