Composants et aspects
si loin, si proche

Jacques Noyé
OBASCO – EMN/INRIA

Une perspective « langage »

- On aura toujours besoin de programmer
- Mais à une autre échelle
- APL : *Architecture Programming Languages*
- Comprendre concrètement les concepts en faisant le lien entre :
  - Les plateformes industrielles
  - La conception pure (ADL, UML)

Il y a trop de concepts !

- Classes/objets
- *Composants*
- *Aspects*
- *Modules*
- Plugin, connecteurs, processus, agents, services web…
- Beaucoup de recouvrement, de variabilité, d'hybrides…

Composants : Industrialiser la production de logiciel [McIlroy68]

- Production de pièces détachées (à qualifier et paramétrer)
- Marché
- Lignes d'assemblages et des lignes de produits
- Adaptation, maintenance et évolution (par remplacement de pièces détachées)

Découplage fort entre producteurs et consommateurs.

Modules : *Information Hiding*
[Parnas72]

- Un système doit être structuré sous forme de modules qui cachent (derrière *des interfaces*) l'information susceptible de changer.
- Le système peut évoluer à partir de modifications locales.
- L'*implémentation* de chaque module ainsi que la vérification de sa correction peut être traitée séparément.

Aspects: Séparation des préoccupations [Dijkstra72]

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that is should be efficient and we can study its efficiency on another day [...]

But nothing is gained on the contrary by tackling these various aspects simultaneously. It is what I sometimes have called ``the separation of concerns" [...]

A scientific discipline separates a fraction of human knowledge from the rest: we have to do so, because, compared with what could be known, we have very, very small heads.

Edsger W. Dijkstra

Composants

- Hybride module et classe/objet
- Module : encapsulation forte (pas d'héritage d'implémentation), interface avec des dépendances explicites
- Classe/objet : existence dynamique, instanciation
- Composition hiérarchique
- Facilités de configuration et d'adaptation

Programmation incrémentale par héritage

```
class Foo {
  void foo() {…}
  void bar() {…}
}
class FooWithProfiling extends Foo {
  static int nbCalls = 0;
  void foo() {
    nbCalls++;
    super.foo();
  }
}
```

En terme d'aspects

- *Programme de base* avec des points d'exécution qui sont des réceptions d'appel.
- La superclasse modifie la sémantique de certains de ces points d'exécutions : *les points de jonction* (réceptions de `foo`).

Avec des mixins

La préoccupation est paramétrée par la classe de base : on voit apparaître la notion de *liaison* entre code de base et aspect.

```
mixin Profiling {
  inherited void foo();
  int nbCalls = 0;

  public void foo() {
    nbCalls++;
    System.out.println("foo " + nbCalls);
    super.foo();
  }
}
class FooWithProfiling = Profiling extends Foo {}
```

Avec AspectJ

La coupe permet de sélectionner les points de jonction.

```
aspect Profiling {
  int nbCalls = 0;

  void around() : call(void Foo.foo()) {
    nbCalls++;
    proceed();
  }
}
```

D'autres coupes

- `call(void Foo.foo()) || call(void Bar.bar())`
- `cflow (call(void Foo.foo())) && call(void Bar.bar())`

Instanciation déclarative

```
aspect Profiling
    pertarget(call(void Foo.foo())) {
  int nbCalls = 0;
  void around():
      call(void Foo.foo()) {
    nbCalls++;
    proceed();
  }
}
```

AOP is Quantification and Obliviousness [FiFr2000]

- Quantification : lors de l'exécution de P, chaque fois que la condition C est vraie, exécuter l'action A.
- Indiscernabilité (*obliviousness / non-invasiveness)* : le programme de base ne fait pas apparaître d'utilisations (au sens *def-use*) des aspects.
- À ne pas confondre avec non anticipation ?

Collaboration

- Notion du folklore de la conception par objets, conceptualisée dans OORASS [Reenskaug92]
- Une application est une composition de *collaborations*.
- Une collaboration est définie par un ensemble d'objets et un *protocole* explicitant comment ces objets interagissent.
- Le protocole définit le *rôle* de chaque objet dans la collaboration.

Exemple : schéma de conception Observateur

```
collaboration ObserverProtocol {
  role Subject {
    provided void addObserver(Observer o);
    provided void removeObserver(Observer o);
    provided void changed();
    expected String getState();
  }
  role Observer {
    expected void notify(Subject s);
  }
}
```

Un air d'interface de module avec des dépendances (*provided/expected*) explicites.

Composition par héritage : les *mixin layers* [CaLi2001]

```
class Obs {                    // couche de base
  public class Subject {...}
  public class Observer {...}
}
class ObsWithProfiling<Observer> // c. paramétrée
    extends Observer{
  public class Subject extends Obs.Subject {...}
  public class Observer extends Obs.Observer
  {...}
}
// application des mixins
ObsWithProfiling<BaseObserverProtocol>
```

Héritage de l'héritage

- Composition asymétrique, linéaire et statique
- Problèmes de robustesse (classe de base fragile) ?
- Langage de liaison limitée (correspondance exacte des noms de méthodes et de classes) nécessite des adaptateurs
- Double utilisation de l'héritage

Composition par agrégation : Caesar [MeOs2003]

- Constat : limitations d'AspectJ (structuration des aspects, déploiement purement statique)
- Aspect :
  - Interface de collaboration (générique)
  - Implémentation (générique)
  - Code de liaison (dépend du programme de base)

Exemple : interface de collaboration

```
interface ObserverProtocol {
  interface Subject {
    provided void addObserver(Observer o);
    provided void removeObserver(Observer o);
    provided void changed();
    expected String getState();
  }
  interface Observer {
    expected void notify(Subject s);
  }
}
```

Implémentation

```
class ObserverProtocolImpl implements
  ObserverProtocol {
  class Subject {
    List Observers = new LinkedList();
    void addObserver(Observer o) {...}
    void removeObserver(Observer o) {...}
    void changed() {
      Iterator it = observers.iterator();
      while (iter.hasNext())
        ((Observer)iter.next()).notify(this);
      }
    }
}
```

Mais, on dirait un composant Java !

- Il en manque un bout :
  - Pas d'implémentation du rôle *Observer* (défini dans le code de liaison)
  - Pas d'appel à la méthode requise `getState` (appelée dans le code de liaison)
- C'est le code de liaison qui fait la spécificité de l'aspect

Code de liaison

Composition par agrégation

```
class ColorObserver binds ObserverProtocol {
  class PointSubject binds Subject wraps Point {
    String getState() {
      return "Point colored "+wrappee.getColor();
    }
  }
  class ScreenObserver binds Observer wraps Screen {
    void notify(Subject s) {
      wrappee.display("Color changed: "+s.getState());
    }
  }
  after(Point p): (call(void p.setColor(Color)))
    { PointSubject(p).changed();}
}
```

Code de liaison

Rajout des méthodes manquantes dans les classes de bases (Point et Screen)

```
class ColorObserver
  class PointSubject
    String getState() {
      return "Point colored "+wrappee.getColor();
    }
  }
  class ScreenObserver binds Observer wraps Screen {
    void notify(Subject s) {
      wrappee.display("Color changed: "+s.getState());
    }
  }
  after(Point p): (call(void p.setColor(Color)))
    { PointSubject(p).changed();}
}
```

Code de liaison

```
class ColorObserver binds ObserverProtocol {
  class PointSubject binds Subject wraps Point {
    String getState() {
      return "Point colored "+wrappee.getColor();
    }
  }
  c                inds Observer wraps Screen {
                   s) {
                   olor changed: "+s.getState());
    }
  }
  after(Point p): (call(void p.setColor(Color)))
    { PointSubject(p).changed();}
}
```

Interception de la méthode à redéfinir

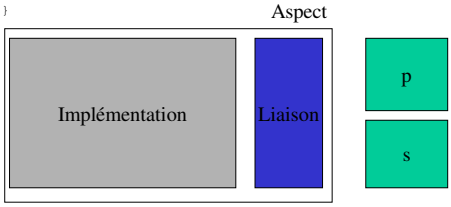
Code de liaison

```
class ColorObserver binds ObserverProtocol {
  class PointSubject binds Subject wraps Point {
    String getState() {
      return "Point colored "+wrappee.getColor();
    }
  }
  class Screen                                  Screen {
    void notif
      wrappee                                   getState());
    }
  }
  after(Point p): (call(void p.setColor(Color)))
    { PointSubject(p).changed();}
}
```

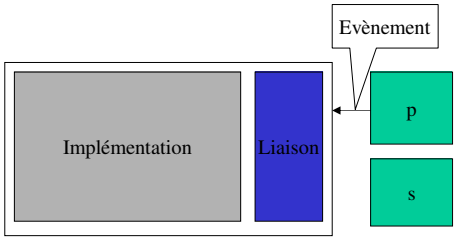
Liaison entre classes du programme de base et de l'aspect

Déploiement

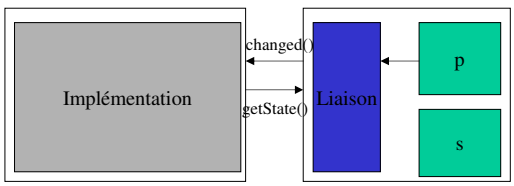
```
deploy class CO extends
  ObserverProtocol<ColorObserver,ObserverProtocolImpl>{};
…
void register(Point p, Screen s) {
  CO.THIS.PointSubject(p).addObserver(
    CO.THIS.ScreenObserver(s));
}
```



Interactions aspect - composant



Interactions composant - composant



Retour aux sources [Kal97]

- *With respect to a system and its implementation using a generalized-procedure language, a property that must be implemented is:*
  - *A component, if it can be cleanly encapsulated in a generalized procedure.*
  - *An aspect if it cannot.*

Composition par tissage : les Aspectual Collaborations

[LoLO03,Ov04

```
collaboration observer_protocol;
participant Subject {
  expected String getState();
  List Observers = new LinkedList();
  void addObserver(Observer o) {...}
  void removeObserver(Observer o) {...}
  aspectual RV changed(CM e) {
    Iterator it = observers.iterator();
    while (iter.hasNext())
      ((Observer)iter.next()).notify(this);
    }
    RV retval = e.invoke();
    return retval;
  }
participant Observer {
  expected void notify(Subject s);
}
```

```
collaboration co;
participant Point {
  expected getColor();
  String getState() {
    return "Point colored "+this.getColor();
  }
}
participant Screen {
  expected void display(Screen);
  void notify(Point p) {
    display("Color changed: "+p.getState());
  }
attach display, observer_protocol {
  Point += Point, Subject {
    around setColor do changed;
  }
  Screen += Screen, Observer {
  }
}
```

Conclusion

- Un nouvel animal dans notre zoo de concepts
- La frontière entre composant et aspect est ténue : le code de liaison
- Essentiellement différents modes d'interaction (fonctionnel ou pas)

Questions ouvertes

- Absence de représentant des collaborations à l'exécution
- Aspects dynamiques
- Aspects et raisonnement modulaire (voir aussi les Open Modules de J. Aldrich)
- Problèmes de typage non triviaux