Conception détaillée

Gerson Sunyé
Université de Nantes
http://sunye.free.fr

Plan

- Introduction
- Différentes approches de conception.
- Heuristiques de conception

Introduction

- Durant l'analyse, nous avons utilisé l'abstraction pour nous débarrasser des détails techniques et simplifier les modèles.
- Cette abstraction était nécessaire pour bien comprendre le domaine, sans se faire influencer par les détails de mise en œuvre.

- Durant la conception détaillée, nous allons faire l'inverse, c’est à dire:
  - Ajouter des détails techniques aux modèles d’analyse.
  - ou alors:
  - Créer un modèle de conception à partir de zéro.

Objectif

- Produire un modèle très proche du code.
- Enrichir les diagrammes plus proches du code (classes, objets).

Approches de Conception

- Conception pour
  - l'extensibilité
  - la performance
  - la portabilité
  - la testabilité

Extensibilité

Concevoir pour l'extensibilté

- Une bonne conception est ouverte.
- elle s'adresse à une classe de problèmes plutôt qu'a un seul problème.
- elle ne doit pas introduire des concepts qui sont encore flous.

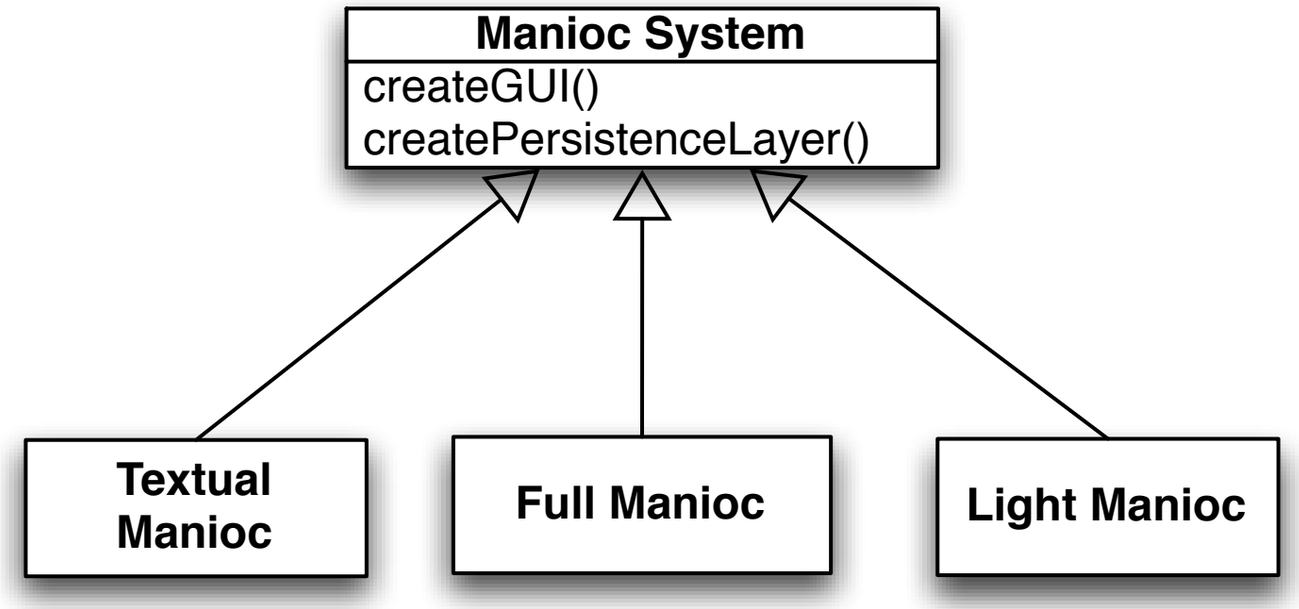
Application de patrons

Ce qui varie	Patron de conception
Algorithme	Stratégie, Visiteur
Actions	Commande
Implémentation	Pont (Bridge)
Réactivité au changement	Observateur
Interaction entre objets	Médiateur
Création d'objets	Fabrique, Prototype
Structure crée	Builder
Algorithme de traversée	Itérateur
Interface des objets	Adaptateur
Comportement de l'objet	Décorateur, Etat (State)

Réifier les variantes

- Appliquer le patron fabrique abstraite pour configurer dynamiquement un des produit d'une famille.

Le système Manioc



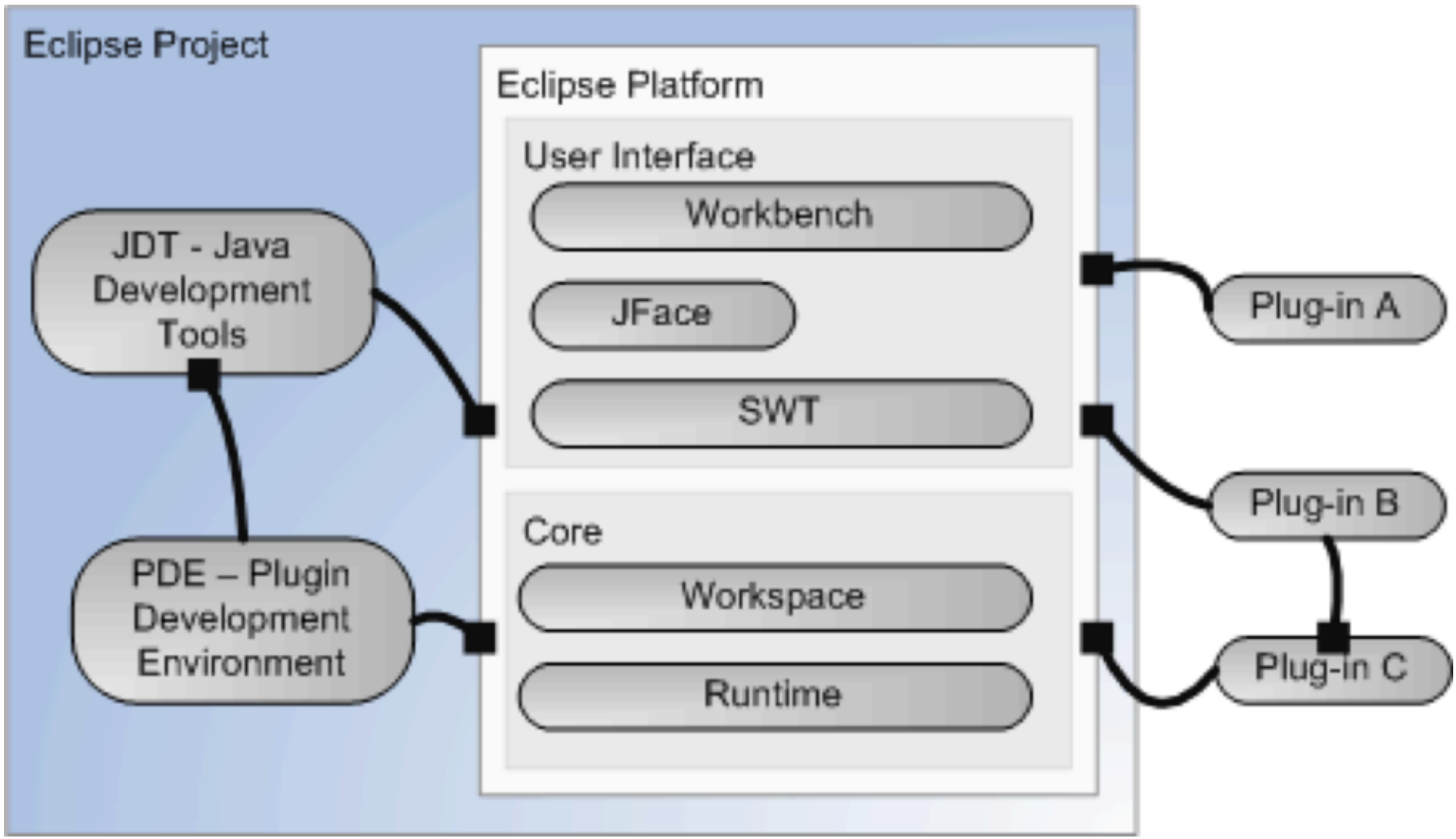
Réifier les états

- Appliquer le patron état sur les diagrammes état-transition

Réifier les événements

- Appliquer le patron commande pour implémenter les événements

Extensibilité
Projet Eclipse



Extensibilité
Projet Eclipse

```
<extension point="org.eclipse.ui.actionSets">
   <actionSet
    id="de.korayguclu.eclipse.actionSet"
    label="Sample Action Set"
    visible="true">
    <menu id="sampleMenu"
          label="Sample &amp;amp;Menu">
     <separator name="sampleGroup"/>
    </menu>
    <action
     id="de.korayguclu.eclipse.actions.SampleAction">
     class="de.korayguclu.eclipse.actions.SampleAction"
     menubarPath="sampleMenu/sampleGroup"
     toolbarPath="sampleGroup"
     label="&amp;amp;Sample Action"
     icon="icons/sample.gif"
     tooltip="Hello, Eclipse world">
    </action>
   </actionSet>
   </extension>
<extension point="org.eclipse.ui.perspectiveExtensions">
    <perspectiveExtension
     targetID="org.eclipse.ui.resourcePerspective">
    <actionSet id="de.korayguclu.eclipse.actionSet"/>
    </perspectiveExtension>
   </extension>
```

Performance

Concevoir pour la performance

- D'abord implémenter correctement, ensuite mesurer la performance et si nécessaire, l'optimiser.
- Maintenir la consistance entre différentes représentations:
  - Vérifier que la version optimisée est équivalente avec la conception.

Performance

- Utiliser, si nécessaire, différents paradigmes de programmation:
  - Créer des interfaces qui séparent deux paradigmes.
  - Ne pas polluer un paradigme avec l'autre côté.
- Exemples: Swig, JNI, P/Invoke.

Règles d'optimisation de code [M. Jackson]

- First Rule of Program Optimization: Don't do it.
- Second Rule of Program Optimization (for experts only!): Don't do it yet."
- Third Rule of Code Optimization: Profile first.

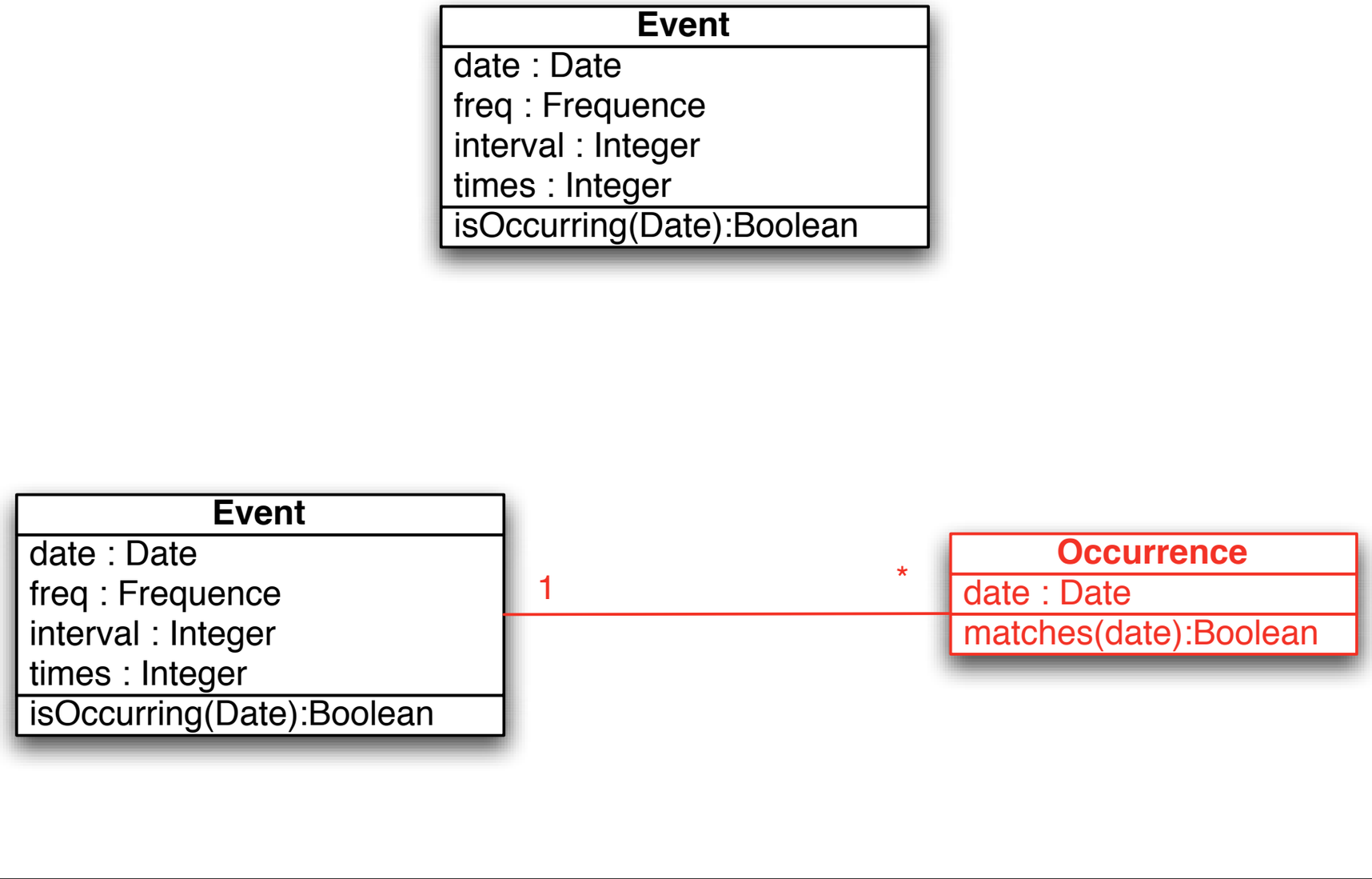
Optimisation d’attributs

- Mémoriser la valeur d’attributs dérivés.

Optimisation d'opérations

- Ajouter des attributs et des associations (éventuellement) redondants pour accélérer un calcul.

Optimisation d’opérations



Portabilité

Concevoir pour la portabilité

- Souvent, traité au niveau du code: utilisation de Java, XUL, Posix, Corba, etc.
- Limites:
  - Même interface pour différents systèmes.
  - Manque d'intégration avec d'autres modules (Agenda, Carnet d'adresses, etc.)
  - Firefox vs. Camino.

Portabilité: solutions

- Développer plusieurs versions d'un même composant (e.g. IHM KDE, IHM Windows, etc.)

Testabilité

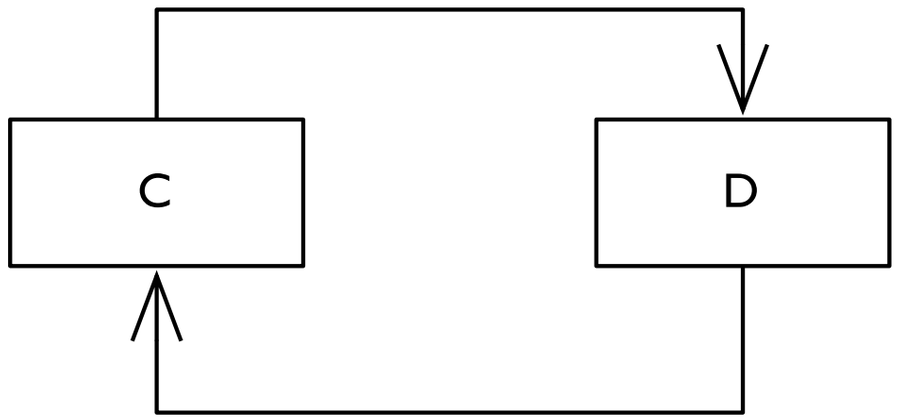
Concevoir pour la testabilité

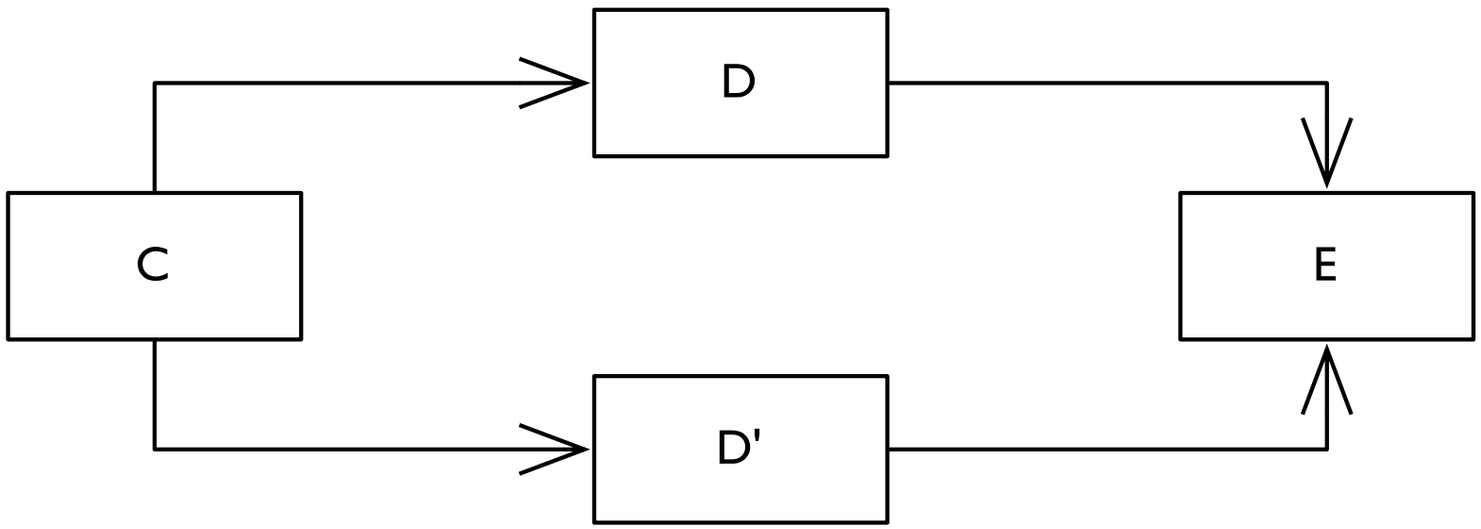
- Enrichir le diagramme de classes avec des dépendances: «uses», «creates», etc.
  - Mise en valeur du couplage entre les classes.
  - Simplification de création d'un plan d'intégration.

Concevoir pour la testabilité

- Donner la préférence à la visibilité publique:
  - les opérations privées ne peuvent pas être testées.
- Utiliser les interfaces
  - elles simplifient le test d'intégration et la création de *mock objects* et *stubs*.
- Eviter les cycles
- Spécifier des Datatypes

Éviter les cycles





Enrichissement du diagramme de classes

Ajustement de l'héritage

- Rechercher la réutilisation : rendre semblable ce qui l'est presque:
  - modification éventuelle de la signature d'opérations
- Abstraction de comportements communs, bibliothèques

Attributs

- Préciser:
  - Valeurs par défaut
  - Multiplicités: [0..1], [*], etc.
  - Contraintes (expressions OCL).
  - Modificateurs: readOnly, redefines, etc.

Rôles

- Préciser les relations entre différents rôles: union, subsets <x>, redefines <y>, ordered, unique, etc.
- Préciser la navigabilité.

Opérations

- Préciser la direction des paramètres.
- Préciser les propriétés des valeurs de retour: readOnly, ordered, etc.

Spécification d'opérations

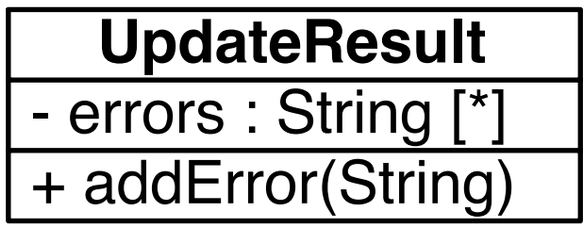
- Expressions OCL
- Action Semantics (xUML)

Traitement d'erreurs

- Traitement d'erreurs:
  - Valeurs de retour.
  - Utilisation d'exceptions.
  - Séparer les erreurs des failles: les dernières sont attendues, les premières, non.

Collecting Parameters [Beck96]

...when you need to collect results over several methods, you should add a parameter to the method and pass an object that will collect the results for you...”



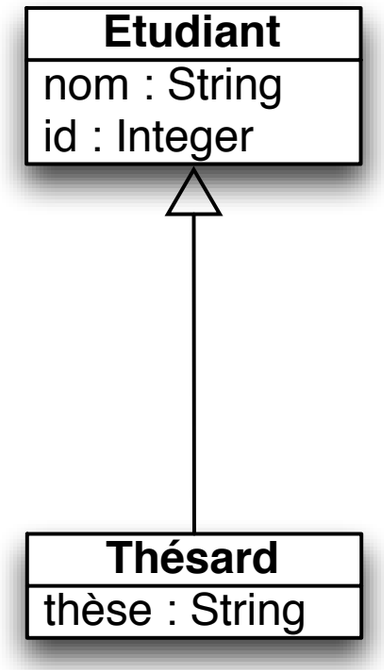
Heuristiques de conception [Riel96]

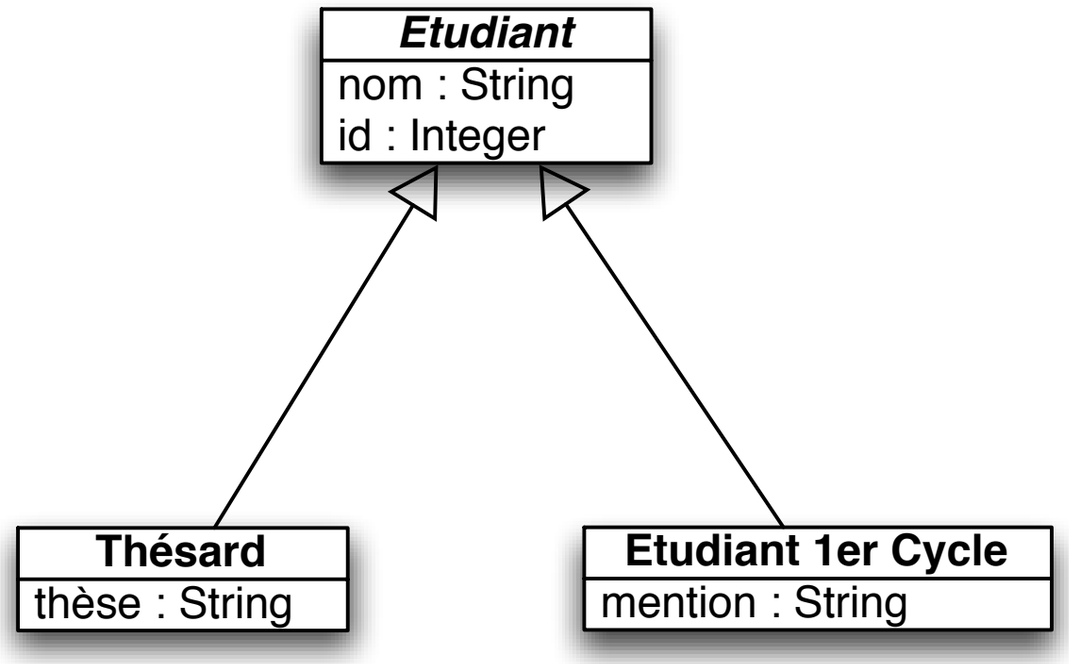
Heuristiques

- Éviter les classes toute puissantes: le système est composé d'une seule classe qui contrôle tout et de plusieurs classes-données.
- Minimiser les relations entre classes.
- Ne pas confondre héritage et agrégation: si le choix est possible, utiliser plutôt l'agrégation.

Spécialisation

- Éviter la spécialisation de classes concrètes
- Les super-classes sont de préférence des classes abstraites.





- Les super-classes ne doivent pas connaître leurs sous-classes.
- Théoriquement, les hiérarchies d'héritage doivent être profondes.
- De façon pragmatique, elles ne doivent pas avoir plus de 6 niveaux.

- Si deux classes partagent les mêmes attributs mais pas le même comportement, ces attributs doivent être placés dans une troisième classe.

Attributs

- Les attributs doivent être cachés à l'intérieur d'une classe : les méthodes Setter/Getter ne sont pas obligatoires
- Utiliser des attributs privés plutôt que des attributs protégés.

Classes

- Les clients d'une classe doivent dépendre seulement de son interface publique. Une classe ne doit pas dépendre de ses clients.
- Implémenter une interface minimale pour toutes les classes: copy(), deepCopy(), equals(), fromString(), toString(), etc.

- Une classe représente une et une seule abstraction.
- Les classes sont des vraies abstractions et non des rôles différents joués par un objet.
- Faire attention aux classes dont le nom est un verbe.

Méthodes

- Les méthodes d'une classe doivent utiliser les attributs (ou les méthodes) de cette classe.
- Les méthodes privées (ou protégées) ne doivent pas modifier l'état d'un objet.

Conclusion

- Séparer interface et implémentation.
- Déterminer ce qui est commun et ce qui est variable.
- Permettre que les implémentations variables soient remplacées grâce à une interface commune.

Conception détaillée