Conception Orientée Objet

Les principes de base

Tianxiao LIU
Master IISC 1ère Année
CY Cergy Paris Université
http://depinfo.u-cergy.fr/~tliu/coo.php

Sommaire

- Introduction au module COO
  - Motivation, programme & organisation, MCC
- UML diagramme de classes
  - Relations entre les classes
  - Comment construire un diagramme de classes
- Les principes de base d'objet
  - OCP, LSP, DIP, ISP, CARP, LoD

Motivation du cours

- Approche objet
  - Incontournable pour le développement des systèmes ***complexes***
  - ***Moins intuitive*** que l'approche fonctionnelle

- Besoins de conception
  - Rigueur pour la compréhension, la comparaison des solutions et la maintenance du système
  - ***Suivre les évolutions incessantes des technologies et des besoins applicatifs***

Motivation du cours

- Pour bien mener la conception
  - Besoin d'un outil de modélisation **unifié**
  - Un outil graphique au lieu de textuel

- UML (*Unified Modeling Language*)
  - Offrir ***plusieurs*** modèles du ***même*** système
  - Chacun met en valeur des aspects différents : fonctionnels, statiques, dynamiques et organisationnels

Programme et organisation

- Approche objet et modélisation UML
  - Principes d'approche objet et design patterns
  - Diagramme de ***classes*** *(class)*
  - Diagramme de ***cas d'utilisation*** *(user case)*
  - Diagramme d'***activités*** *(activity)*
  - Diagramme de ***séquence*** *(sequence)*
  - Diagramme d'***états-transitions*** (*statechart*)
- Architectures des logiciels
- *Domain Driven Design (DDD)*

Programme et organisation

- Technologies Java SE / Java EE
  - Java **multi-threading**
  - **JDBC** (*Java Database Connectivity*)
  - **Hibernate** ORM framework + JPA Annotation
  - **Spring** Framework (IoC + AOP)
  - **JSF** (*Java Server Faces*) Web framework (MVC)
- Outils technologies Java
  - Java 1.6 ou plus + les librairies frameworks
  - Eclipse pour Java EE
  - MySQL ou PostgreSQL
  - Tomcat : serveur Web

MCC de l'UE COO

- Partie examen (50%)
  - Tous documents autorisés (sauf livres et appareils électroniques)
  - Principes, patterns, modélisation UML
  - Architectures des logiciels et DDD
- Partie CC (50%)
  - Projet transversal – Janvier 2024
    - Atelier Gestion de Projet (AGP) + COO + BDA
    - Evaluation de la conception et de la réalisation

Diagramme de classes UML (cd)

- Une vue interne du système à concevoir
  - Une description purement **statique**
  - Pas de d'indication qui montre comment une opération est réalisée

- Un ensemble de classeurs
  - Classe, classe abstraite, interface, paquetages, objets
  - Les relations entre ces éléments

Notions de base

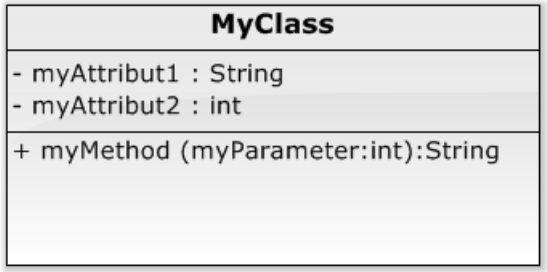
- Objet, classe et instance
  - Un **objet** représente une chose **naturelle** qui a des caractéristiques et des comportements
  - Une **classe** est le regroupement des données et des traitements applicables aux objets qu'elle représente
  - Une **instance** peut être un objet d'une classe, et peut aussi être d'autres éléments du langage UML, ex. un lien est une instance d'une association

Notions de base

- **Interface**
  - Une classe abstraite particulière
  - Motivation
    - Classer les opérations en catégorie sans préciser la manière dont elles sont implémentées
    - Préciser surtout les conditions et les effets de l'invocations des opérations définies dedans
  - Une interface n'a pas d'instances directes
    - peut être réalisée par différentes classes en Java

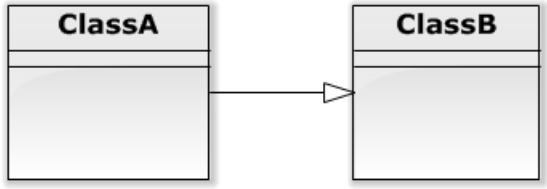
Notions de base

- **Composition d'une classe**
  - attribut
    - visibilité nom : type
  - méthode (opération)
    - visibilité nom (paramètres) : type de retour
    - paramètre = nom : type
  - classes ou méthodes abstraites
    - En italique

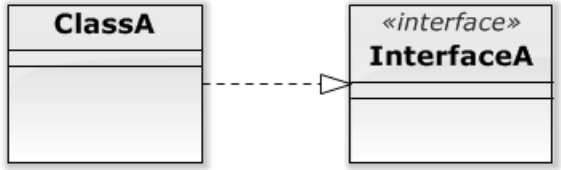


Relations entre classes

- **Héritage**
  - Une classe est la sous-classe (classe fille) d'une autre classe
  - Pas de multi-héritage en Java

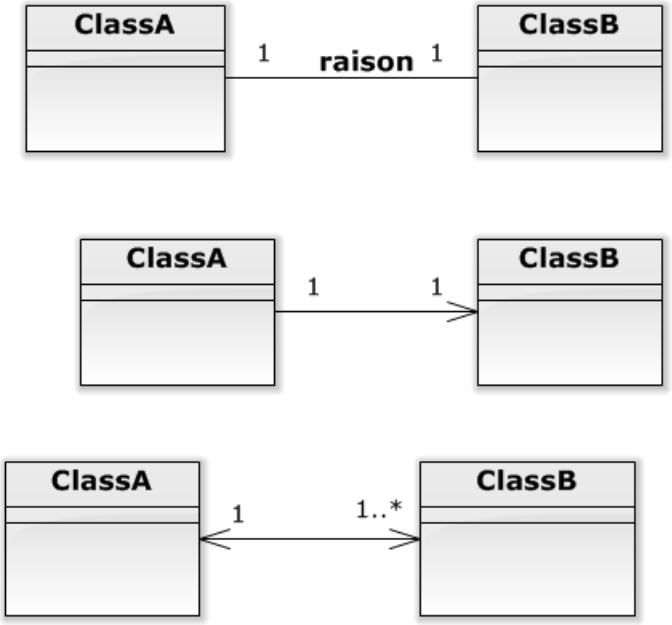


- Réalisation
  - Une classe implémente une interface



Relations entre classes

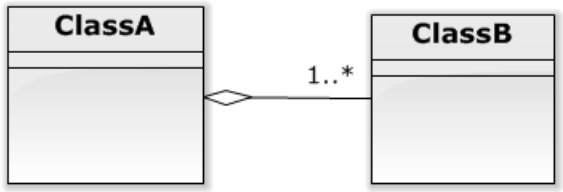
- **Association**
  - Une relation structurelle entre les deux classes
  - **Multiplicité** (ou cardinalité) : nombre d'objets intervenant
    - 1, 1..*, 1..5, …
  - **Nom de l'association**
    - un verbe en actif/passif
  - **Rôles**
    - Un nom
  - **Navigabilité**
    - Association unidirectionnelle
    - Association bidirectionnelle



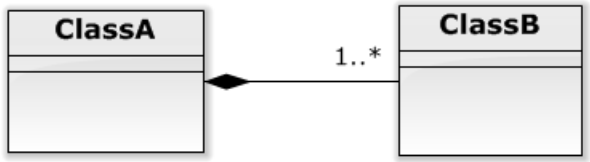
Relations entre classes

- **Agrégation**
  - une forme particulière de l'association
  - Relation d'inclusion structurelle

    ou comportementale



- **Composition**
  - Une agrégation particulière
  - Classe composite et classe composant
  - La classe composite est responsable de la création, de la copie et de la destruction de ses composants



Relations entre classes

- **Dépendance**
  - Relation unidirectionnelle
  - Classe cible et classe source
  - Si modification de la cible, alors changement de la source
  - **Stéréotypes : exemples typiques**
    - **use** : le fonctionnement de la source a besoin de l'utilisation de la cible
    - **call** : appel d'une opération par une autre
    - **create** : une opération de la source crée une instance de la cible

Construction d'un cd

- **Processus de construction**

Construction d'un cd

- Identifier les classes du domaine
  - Idéalement, il faut consulter auprès d'un expert du domaine
  - Faire une liste des classes candidates et éliminez celles redondantes et superflues
  - Définir avec grand soin le niveau de granularité
  - Evitez les implémentations trop détaillées

Construction d'un cd

- Définir les relations entre classes
  - Eviter les chemins redondants (surtout les cercles entre les classes)
  - Pas de relations *insensées*
  - Déterminer les multiplicités (cardinalités) dès que possible
  - En général, pas de classes isolées

Construction d'un cd

- Trouver les attributs des classes
  - N'espérez pas trouver tous les attributs dès la construction du diagramme de classes

  → ***certains attributs seront ajoutés à l'implémentation***
  - Profitez bien de l'héritage, classes abstraites, interfaces, et polymorphisme pour simplifier le diagramme
  - Pas besoin d'illustrer les *getter/setter, constructor, toString*

Bon système VS Mauvais système

- Un bon système
  - Support de maintenabilité + réutilisabilité
  - Maintenabilité ≠ réutilisabilité (niveaux différents)

- Un mauvais système
  - Expression des besoins ☺
  - Conception et réalisation ☺ ☺
  - Livraison : on est content ☺ ☺ ☺
  - Maintenance ? bugs découverts ☹
  - Extension ? très difficile, très chère ☹ ☹
  - Réutiliser le système ☹ ☹ ☹
  - Abandon → ***Recommencer le même cycle ?!!***

Maintenabilité et réutilisabilité

- Réutilisabilité du système
  - Façons traditionnelles
    - Copier-coller du code source
    - Réutiliser les algorithmes
    - Réutiliser les structures de données
  - Approche orientée objet
    - Niveau plus élevé grâce aux aspects objet: abstraction, encapsulation, héritage, polymorphisme
    - On se concentre plus sur la partie **métier** (*business*)
    - Les **design patterns**

Maintenabilité et réutilisabilité

- Extensibilité du système
  - Utilisation correcte des interfaces et des classes abstraites
  - Principes de conception pour extensibilité
    - *Open-Closed Principle (OCP)*
    - *Liskov Substitution Principle (LSP)*
    - *Dependency Inversion Principle (DIP)*
    - *Interface Segregation Principle (ISP)*
    - *Composition/Aggregation Reuse Principle (CARP)*
    - *Law of Demeter (LoD)*

Principe ouvert / fermé (OCP)

- Principe
  - Un système doit s'ouvrir à l'extension et se fermer à la modification
- Réalisation
  - Il faut toujours une très bonne abstraction
  - **EVP** : *Principle of Encapsulation of Variation*
    - Si un élément varie dans le système, encapsulez-le
    - Les variations doivent être encapsulées dans un objet au lieu d'être dispersées partout.
    - Ne pas mélanger deux types de variations → ***héritage ne dépasse généralement pas 2 niveaux***

Principe de substitution de Liskov (LSP)

- Définition
  - Si **q(x)** est une propriété démontrable pour tout objet **x** de type **T**, alors **q(y)** est vraie pour tout objet **y** de type **S** tel que **S** est un sous-type de **T**.

- Principe
  - *method (Base b)* → *method (Derived b)*
  - **L'inverse n'est pas vrai.**
  - Notion de base pour la réutilisation de l'héritage
  - Sous-type indétectable

Principe d'inversion de dépendance (DIP)

- Motivation
  - Problème de dépendance (top-down)
- Solution
  - Utilisation d'une couche d'abstraction
  - Inverser la dépendance : concret ----> abstract
- Trois types de couplages
  - Couplage nul : pas de couplage
  - Couplage concret : référence directe
  - *Couplage abstrait : référence indirecte via classes abstraites ou interfaces*

Principe de ségrégation des interfaces (ISP)

- Principe
  - Diviser une grande interface en plusieurs petites interfaces : hiérarchiser les interfaces
  - Exposer aux clients uniquement ce dont ils ont besoin : ***Customized Service***

- Contamination des interfaces
  - Economiser le nombre d'interface ?

    → Très mauvaise idée

Principe de réutilisation des composites /agrégations (CARP)

- Principe
  - Utiliser un objet d'une classe existante dans une nouvelle classe → réutilisation
  - Préférer composites/agrégation à héritage
  - Différence entre "Has-A" et "Is-A"
  - Composite ici n'est pas le pattern *Composite*

Loi de Déméter (LoD)

- Motivation
  - ***Only talk to your immediate friends***
  - ***Don't talk to strangers***
  - ***Least Knowledge Principle (LKP)***

- LoD entre deux classes
  - Si ces deux classes n'ont pas besoin de se communiquer directement, on utilisera une troisième classe entre les deux pour assurer la communication

- LoD générale
  - Couplage faible entre les modules d'un système
  - Restreindre les accessibilités