Analyse, conception et validation de logiciels

2. UML : concepts, aspects statiques

Mathias Péron

ENSIMAG 2A - année 2005-2006

INP Grenoble

Introduction

- UML (*Unified Modeling Language*) est un langage de modélisation à objets.
- UML permet de visualiser, spécifier, construire et documenter les différentes parties d'un système logiciel.
- UML est une notation graphique basée sur neuf diagrammes.
- UML n'est pas une méthode.
- UML peut être utilisé dans tout le cycle de développement du logiciel, indépendamment de la méthode.

Historique 1/3

De la programmation objet (à l'origine des approches objet)

1967	Simula introduit la notion de classe pour implémenter des types abstraits
1976	Smalltalk introduit les principaux concepts de programmation objet (encapsulation, agrégation, héritage)
1983	C++
1986	Eiffel
1995	Java, Ada 95
2000	C#

Historique 2/3

Des Méthodes

1970	Premières méthodes fonctionnelles
1980	Approches systémiques : modélisation des données et des traitements (Merise)
1990 /95	Apparition d'une cinquantaine de méthodes objets (Booch, OMT, OOA, OOD, HOOD, OOSE . . .)

Historique d'UML 3/3

1989	OMG (*Object Management Group*) : Organisation internationale (plus de 800 membres (informaticiens et utilisateurs)) créée pour promouvoir la théorie et la pratique de la technologie objet dans le développement de logiciel.
1995	Booch (auteur de la méthode Booch), Rumbaugh (auteur de la méthode OMT) et Jackobson (auteur de la méthode OOSE, et des cas d'utilisation) commencent à travailler sur la méthode unifiée (*Unified Method*).
1996	Création d'un consortium de partenaires pour travailler sur la définition d'UML dans l'OMG. Participants : Rationale, IBM, HP, Microsoft, Oracle ...
1997	UML 1.1 (normalisé)
1998	UML 1.2
1999	UML 1.3
2001	UML 1.4
2003	UML 1.5

Objectifs d'UML

Buts des concepteurs d'UML :

- représenter des systèmes entiers (pas uniquement logiciels) par des concepts objets
- lier explicitement des concepts et le code qui les implantent
- pouvoir modéliser des systèmes à différents niveaux de granularité, (pour appréhender des systèmes complexes)
- créer un langage de modélisation utilisable à la fois par les humains et les machines.

UML permet de modéliser les différents aspects d'un système :

- les aspects fonctionnels (fonctionnalités)
- les aspects statiques (classes, relations, objets)
- les aspects dynamiques (états et comportement des objets)

Concepts objets : modélisation & prog. objet 1/2

Idée principale de l'approche objet :

Centraliser les données et les traitements associés dans une même unité, appelée objet.

Objet Un objet est caractérisé par

- une identité (un nom)
- un état, défini par un ensemble de valeurs d'attributs
- un comportement, défini par un ensemble de méthodes

Classe Abstraction qui représente un ensemble d'objets de même nature (*i.e.* ayant les mêmes attributs et méthodes)

- un objet est une « instance » de sa classe

Encapsulation Consiste en la possibilité de masquer les détails d'implémentation (constituants privés d'un objet)

Concepts objets : modélisation & prog. objet 2/2

Agrégation Possibilité de définir des objets composites, fabriqués à partir d'objets plus simples.

Extension Possibilité de définir une nouvelle classe (classe « dérivée ») à partir d'une classe existante. On peut ajouter des attributs ou des méthodes.

- L'extension permet de définir des hiérarchies de classes.

Héritage Une classe dérivée hérite des attributs et méthodes de la classe mère.

- L'héritage évite la duplication de constituants (attributs, méthodes)
- L'héritage encourage la réutilisation

Concepts objets : prog. objet uniquement

Redéfinition d'opérations héritées Dans une classe dérivée, certaines méthodes héritées peuvent être redéfinies (ou spécialisées).

Liaison dynamique L'exécution d'une méthode dépend du type dynamique (type à l'exécution) d'un objet.
Intérêts de ce mécanisme :

- Pouvoir définir de façon uniforme des opérations qui s'appliquent à des objets de types différents, mais dérivés d'une classe commune. (polymorhisme)
- Réutilisation de code existant.

Concepts objets : exemple héritage 1/3

```
/**
 *  Classe des points en deux dimensions.
 */
class Point2D {
   int x, y ; // Abscisse et ordonnée
   /** Constructeur */
   Point2D(int x, int y) {
      this.x = x ;
      this.y = y ;
   }
   /** Distance à l'origine d'un Point2D. */
   double distanceOrigine() {
      return Math.sqrt(x*x +y*y) ;
   }
}
```

Concepts objets : exemple héritage 2/3

```
/**
 *  Dispersion d'un nuage de points.
 */
class Dispersion {
   private Dispersion() { }
   /** Calcul de la dispersion. */
   static double calc(Point2D[] tab) {
      double res = 0 ;
      for (int i = 0; i < tab.length; i++) {
         res += tab[i].distanceOrigine() ;
      }
      return res ;
   }
}

Point2D p1 = new Point2D(3, 4) ;
Point2D p2 = new Point2D(0, 1) ;
Point2D[] tab = {p1, p2} ;
double dispersion = Dispersion.calc(tab) ;
```

Concepts objets : exemple héritage 3/3

```
/**
 * Classe des points colorés en dim. 2.
 */

class Point2DColore extends Points2D {

   int couleur ; // Couleur

   /** Constructeur */
   Point2DColore(int x, int y, int couleur){
      super(x, y) ;
      this.couleur = couleur ;
   }
}

Point2DColore c1 = new Point2DColore(3, 4, 10) ;
Point2DColore c2 = new Point2DColore(0, 1, 20) ;
Point2DColore[] tab2 = {c1, c2} ;
double disp2 = Dispersion.calc(tab2) ;
```

Concepts objets : exemple liaison dynamique 1/3

```
/**
 * Classe des points en trois dimensions.
 */
class Point3D extends Point2D {
   int z ; // Hauteur
   /** Constructeur */
   Point3D(int x, int y, int z) {
      super(x, y) ;
      this.z = z ;
   }
   /** Distance à l'origine. */
   double distanceOrigine() {
      return Math.sqrt(x*x + y*y + z*z) ;
   }
}

Point3D q1 = new Point3D(0, 3, 4) ;
Point3D q2 = new Point3D(0, 0, 1) ;
Point3D[] tab3 = {q1, q2} ;
double disp3 = Dispersion.calc(tab3) ;      // -> 6
```

Aspects fonctionnels 1/4

On utilise le diagramme de cas d'utilisation.

Acteur Rôle joué par une personne ou une chose qui interagit avec le système.

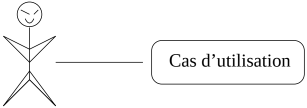
- Un acteur peut être une personne, un paramètre de l'environnement (comme une grandeur physique), ou un autre système.

Cas d'utilisation Manière spécifique d'utiliser le système. Image d'une fonctionnalité du système.

- Le diagramme de cas d'utilisation permet de montrer les interactions entre les acteurs et les différentes fonctionnalités du système.

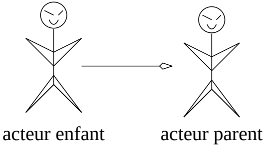
Notations

- Relation entre un acteur et un cas d'utilisation :



  L'acteur « déclenche » un cas d'utilisation.

- Généralisation entre deux acteurs :

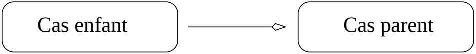


  Toute instance de l'acteur enfant est une instance de l'acteur parent. Tout ce que peut réaliser l'acteur parent, l'acteur enfant peut le réaliser.

Aspects fonctionnels 3/4

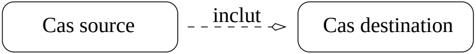
Notations (suite)

- Généralisation entre deux cas d'utilisation :



  Le cas enfant spécialise le cas parent.

- Inclusion entre deux cas d'utilisation :



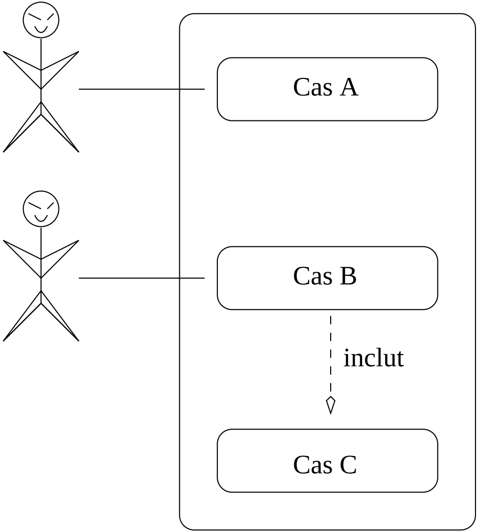
  Les comportements décrits par le cas source incluent les comportements décrits par le cas destination.

- Relation d'extension entre deux cas d'utilisation

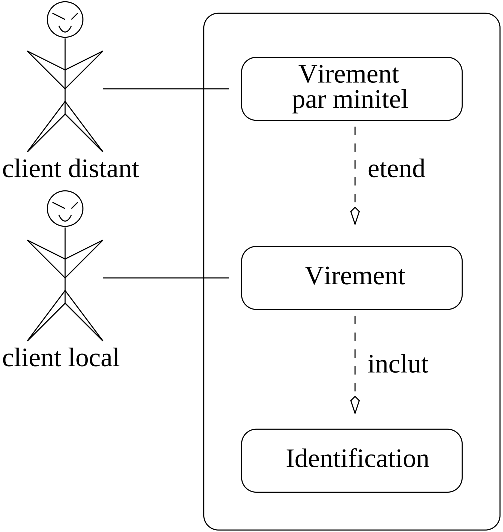
  Les comportements décrits par le cas source étendent les comportements décrits par le cas destination.

Aspects fonctionnels 4/4

- Limites du système



Système de virements pour une banque



Aspects fonctionnels : exemple 2

Distributeur de billets