IPRO

CONCEPTION ORIENTÉE OBJET : QUELQUES PRINCIPES & OUTILS

Guillaume Bouyer

ENSIIE

www.ensiie.fr/~bouyer

CONTENU DU COURS

INTRODUCTION À UML

UML

- Unified Modeling Language
- Langage visuel dédié à la spécification, la construction et la documentation des modèles d'un système logiciel
  - Notation (graphique) standard pour la modélisation d'applications à base d'objets ou de composants
  - Complété par un autre langage textuel pour exprimer des contraintes : OCL
  - Version 1.0 en 1997, version 2.5 en 2015
- Créé par l'Object Management Group (OMG)
  - Fondé en 1989 pour standardiser et promouvoir l'objet

DIFFÉRENTS USAGES D'UML [FOWLER 2003]

1. *Esquisse* (cf méthodes Agile) :
   - Diagrammes informels et incomplets
   - Support pour aider à la communication (client...), faciliter la compréhension d'un système, concevoir les parties critiques
2. *Plan* :
   - Diagrammes formels relativement détaillés
   - Annotations en langue naturelle
   - Génération autom. possible d'un squelette de code
3. *Programmation* (Model Driven Architecture/MDA) :
   - Spécification complète et formelle
   - Pour des applications bien particulières, génération autom. d'un exécutable

DIFFÉRENTS MODÈLES UML

Différents points de vue sur un système :

- Modèles descriptifs : décrit l'existant (domaine, métier)
- Modèles prescriptifs : décrit le futur système à réaliser
- Modèles pour l'utilisateur : quoi
- Modèles pour les concepteurs/développeurs : comment
- Modèles statiques : aspects structurels
- Modèles dynamiques : comportements et interactions

DIFFÉRENTS TYPES DE DIAGRAMMES

Différents outils de modélisation

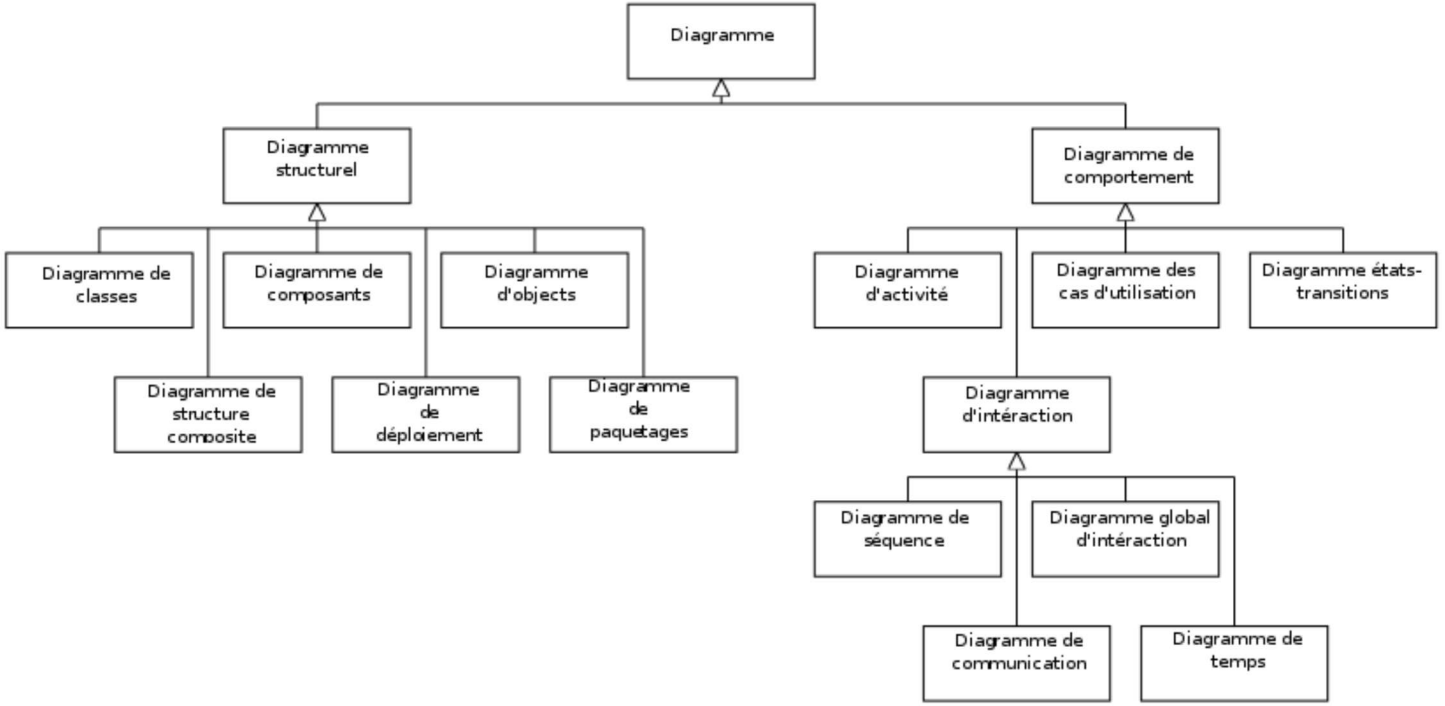


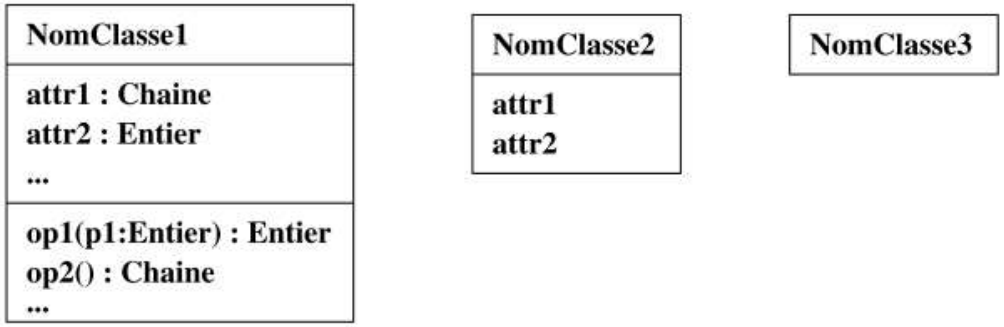
DIAGRAMME DE CLASSE

Structure statique d'un système sous forme de graphe

Représentation des classes et de leurs relations

REPRÉSENTATION DES CLASSES

Différents niveaux de détail :



- Nom
- Attributs : type, valeurs par défaut, visibilité, caractéristiques…
- Opérations : signature, visibilité, caractéristiques…

REPRÉSENTATION DES ATTRIBUTS

Forme simple :

```
<visibility> nomAttribut : <type>
avec
  type = primitif ou classe
  visibility = + (public), # (protected), - (private), ~ (package)
```

Autres options :

```
<visibility> nomAttribut[multiple] : <type> = <initial value> {property}
  avec multiple = nombre d'éléments ou min .. max
  si attribut statique : souligné
```

Exemples

```
- x : Real = 10.0
+ coordinates[3] : Real
subscribers[2..8] : People
- balance : Dollars
- timeOfFlight : Minutes {not null}
# annualRate : Percentage
```

REPRÉSENTATION DES OPÉRATIONS

```
<visibility> nameOperation (<arguments>) : <returnType>
avec
  arguments selon le format : name : type = <defaultValue>`
```

Méthode de classe (`static`) : nameOperation()

MEMBRES DÉRIVÉS

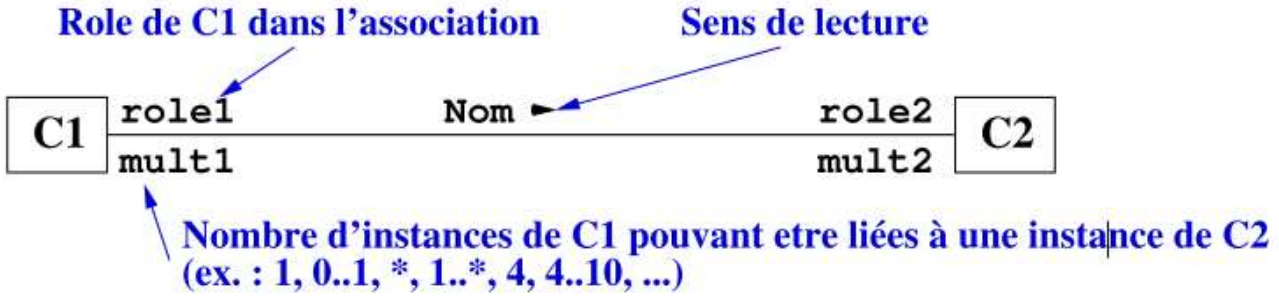
- Propriété redondante, dérivée d'autres propriétés déjà allouées
- Notation précédée de / : /nomAttribut
  - ex : Classe Boule avec attributs rayon, PI et /volume
- Un attribut dérivé peut donner lieu à une opération qui encapsulera le calcul effectué

ASSOCIATION ENTRE CLASSES

Exprime une connexion sémantique entre des classes (le plus souvent 2)

- **Nom : décrit la relation**
  forme verbale, active ou passive
  avec sens de lecture
- **Rôles : décrivent comment chaque classe voit l'autre**
  forme nominale
- **Arités : portent une indication de multiplicité**
  nombre d'objets de la classe considérée pouvant être liés à un objet de l'autre classe

ASSOCIATION : INTERPRÉTATION



- une classe C1 “nom” une classe C2 (ou l'inverse selon sens)
- une classe C1 est “rôle1” d'une classe C2 et “mult1” C1 peuvent jouer ce rôle pour une C2
- la classe C1 a un attribut de nom “rôle2” dont le type est C2 si “mult2” ∈ {1, 0..1} ou collection de C2 sinon

ASSOCIATION : ARITÉS

1	un et un seul
0..1	zéro ou un
M .. N	de M à N (entiers naturels)
*	de zéro à plusieurs
0 .. *	de zéro à plusieurs
1 .. *	de un à plusieurs

« Un JEU comporte plusieurs joueurs »

« Un JOUEUR joue dans un seul jeu »

ASSOCIATION : NAVIGABILITÉ

- Capacité d'une instance de C1 (resp. C2) à accéder aux instances de C2 (resp. C1)
- par défaut : navigabilité dans les deux sens
  - C1 a un attribut de type C2 et C2 a un attribut de type C1
- peut être spécifiée/orientée
  - ex : C1 a un attribut du type de C2, mais pas l'inverse

ASSOCIATION : COMPOSITION

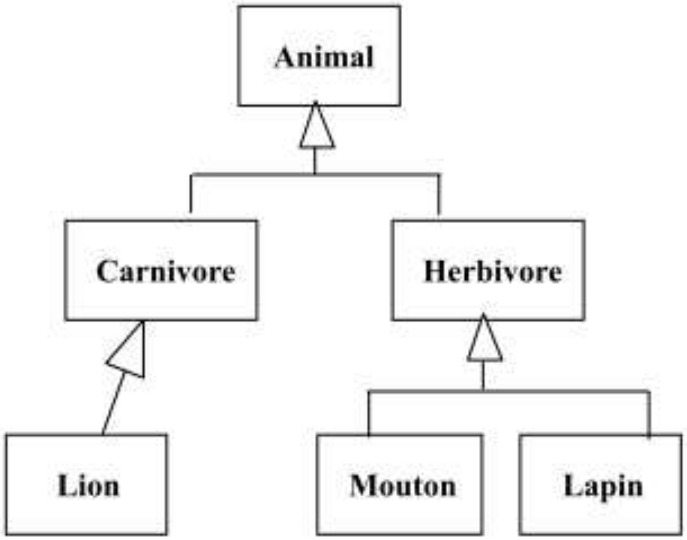
- Attributs contenus physiquement dans la classe composite
- Relation transitive et antisymétrique
- La création (resp copie, destruction) du composite (container) implique la création (resp copie, destruction) de ses composants
- Un composant appartient à au plus un composite

ASSOCIATION : AGRÉGATION

- Simple regroupement de parties dans un tout
- La création (resp. la destruction) des composants est indépendante de la création (resp. la destruction) du composite
- Un objet peut faire partie de plusieurs composites à la fois

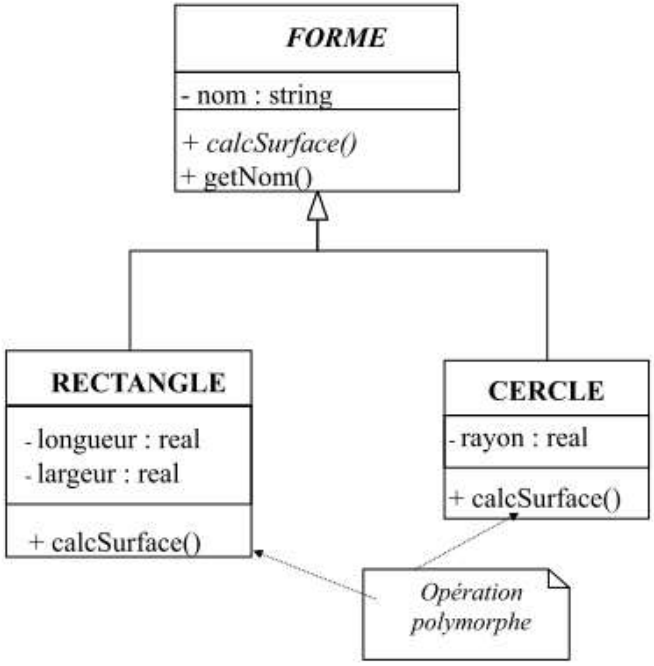
HÉRITAGE/GÉNÉRALISATION/SPÉCIALISATION

- Relation transitive, non réflexive, et non symétrique
- Rappel : la sous-classe “est-une-sorte-de” la super classe
  - Toute instance de la sous-classe est instance de la super classe



CLASSES ET OPÉRATIONS ABSTRAITES

- Classe : mot clé {abstract} devant le nom, ou nom en *italique*
- Opération : nom en *italique*



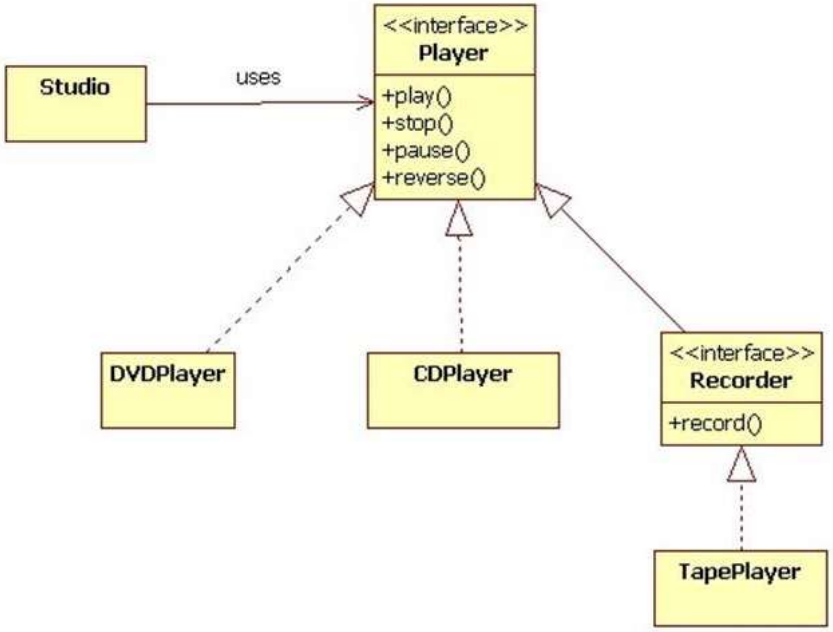
INTERFACE

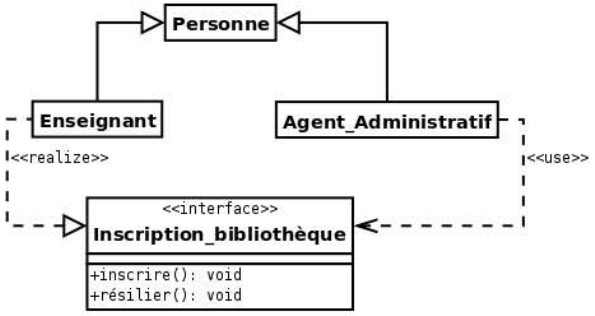
Mot clé «interface» devant le nom

Héritage : identique à l'héritage entre classes

Implémentation : pointillés

Utilisation : pointillés et mot clé «use»





(b)

DIAGRAMME DE CLASSE : EXEMPLE RÉCAPITULATIF

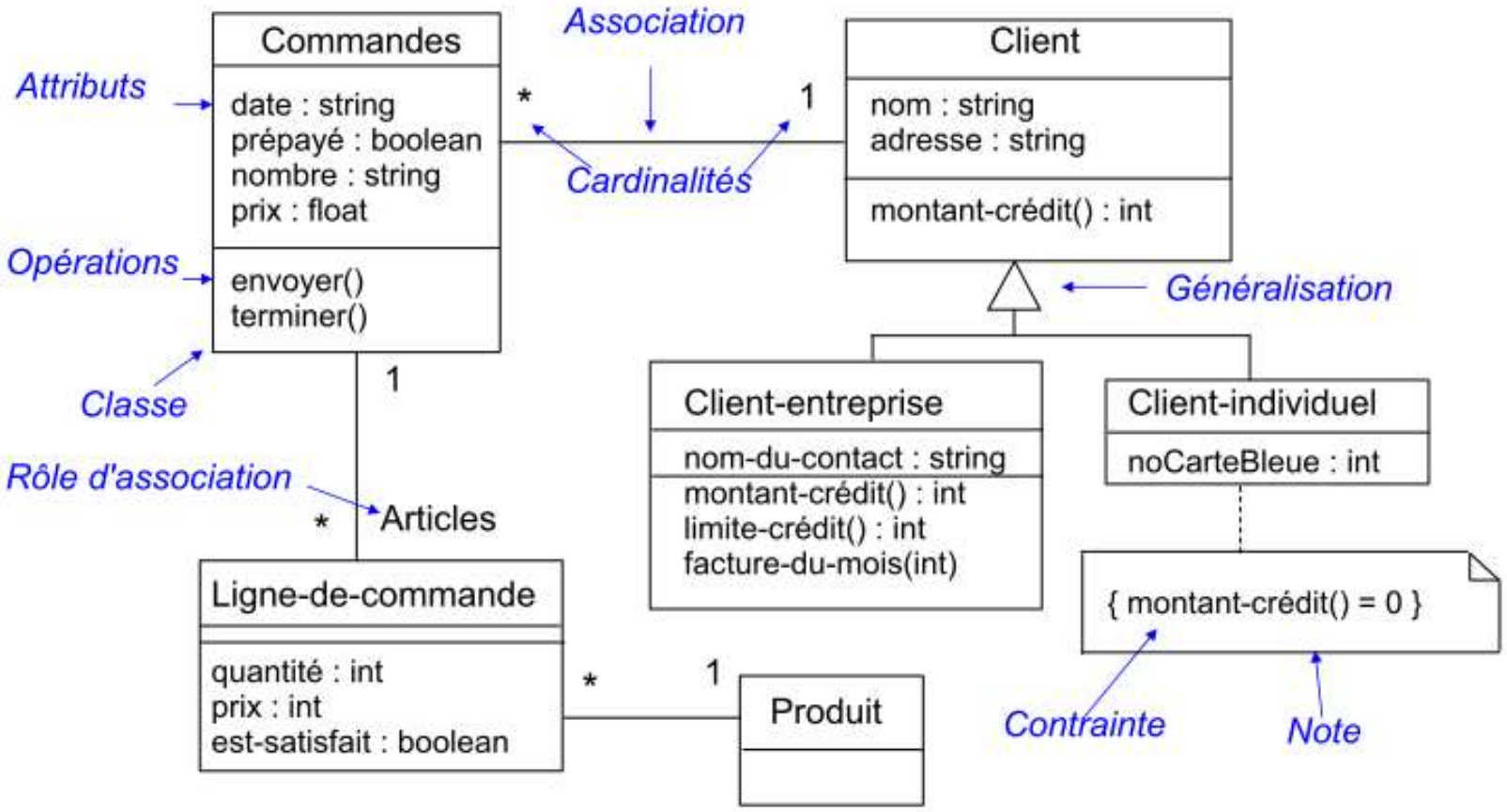


DIAGRAMME DES CAS D'UTILISATION

USE-CASE DIAGRAMS (UCD)

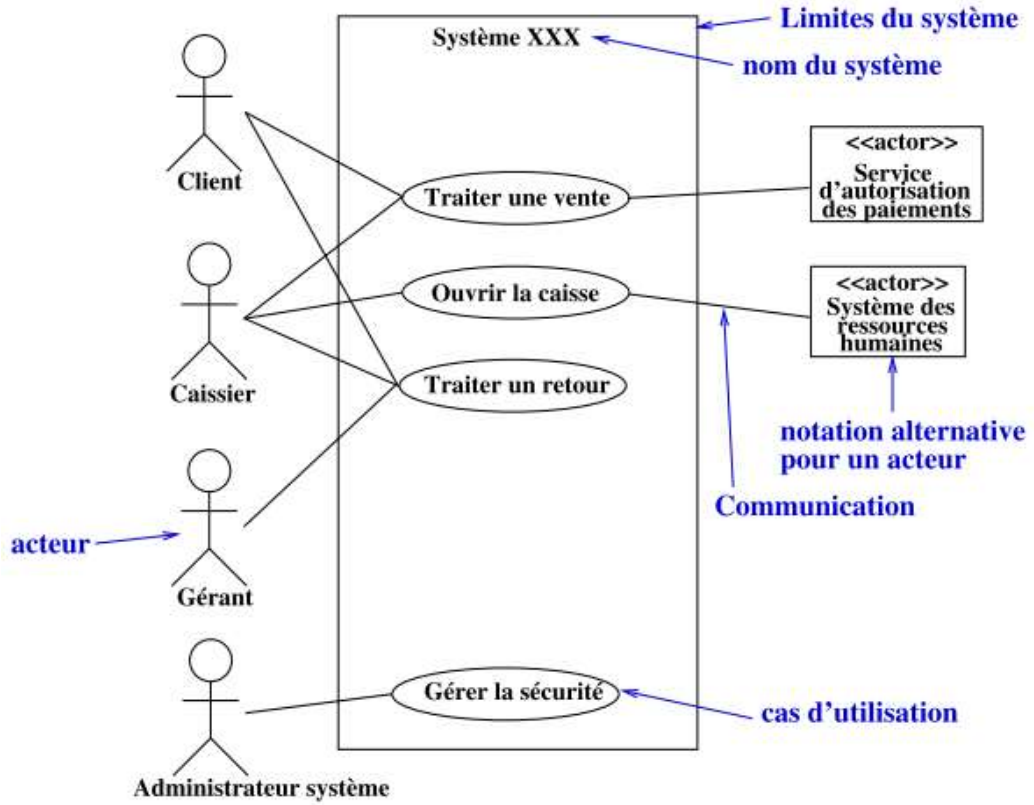


DIAGRAMME DES CAS D'UTILISATION : OBJECTIFS

- Permettre au client/utilisateur de décrire ses besoins
  - Déterminer les buts et l'étendue du système
  - Support de communication
  - Aide à un accord (contrat) entre clients et développeurs
- Aide au développement
  - Support de communication
  - Découpage en tâches réparties
  - Concevoir les IHM
  - Base pour les tests fonctionnels

“CAS D’UTILISATION”

- Expression du comportement du système du point de vue de l'utilisateur
- Séquence d’interactions entre le système et un ensemble d’“acteurs”
- Représentation générale et synthétique d'un ensemble de scénarios similaires
  - *ex :* “un joueur joue un coup”

LES ACTEURS

- Interagissent avec le système, en échangeant de l'information (en entrée et en sortie)
  - personne ou autre système
  - utilisateurs directs
  - ou qui fournissent un service au système
- Ne pas raisonner en terme d'entité physique, mais de rôle
  - Une même entité peut jouer le rôle de plusieurs acteurs
  - Plusieurs entités peuvent jouer le même rôle, et donc agir comme le même acteur
- Définir les cas d'utilisation consiste à identifier les buts des acteurs (verbe à l'infinitif)

RELATION ENTRE CAS D'UTILISATION

- Généralisation
- Inclusion
- Extension

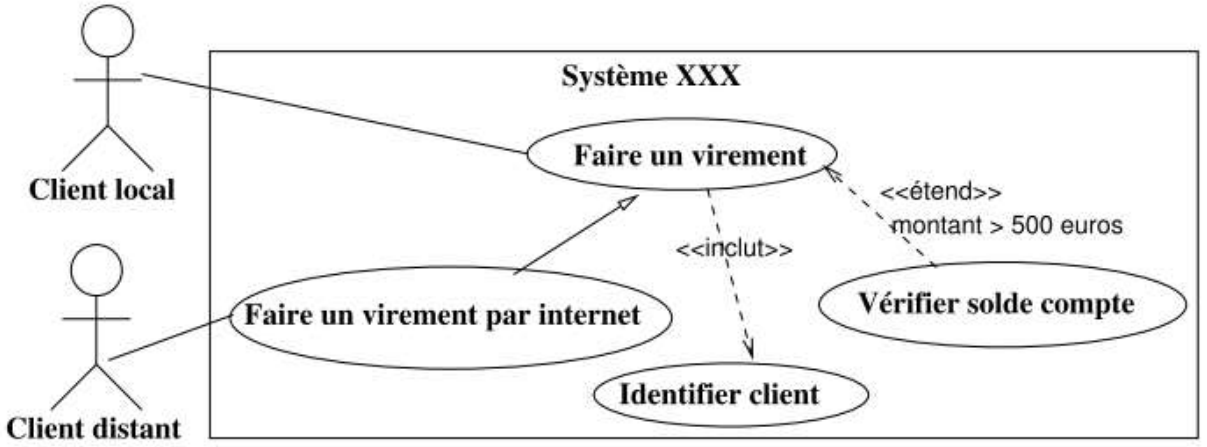


DIAGRAMME DE SÉQUENCE

Représentation graphique d'un scénario particulier

- des objets dans une situation donnée (instances)
- les messages échangés entre les objets

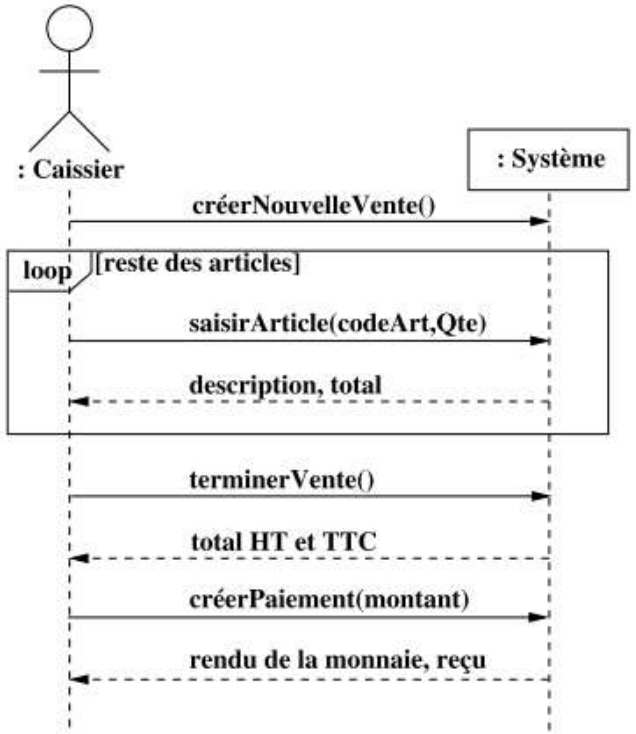
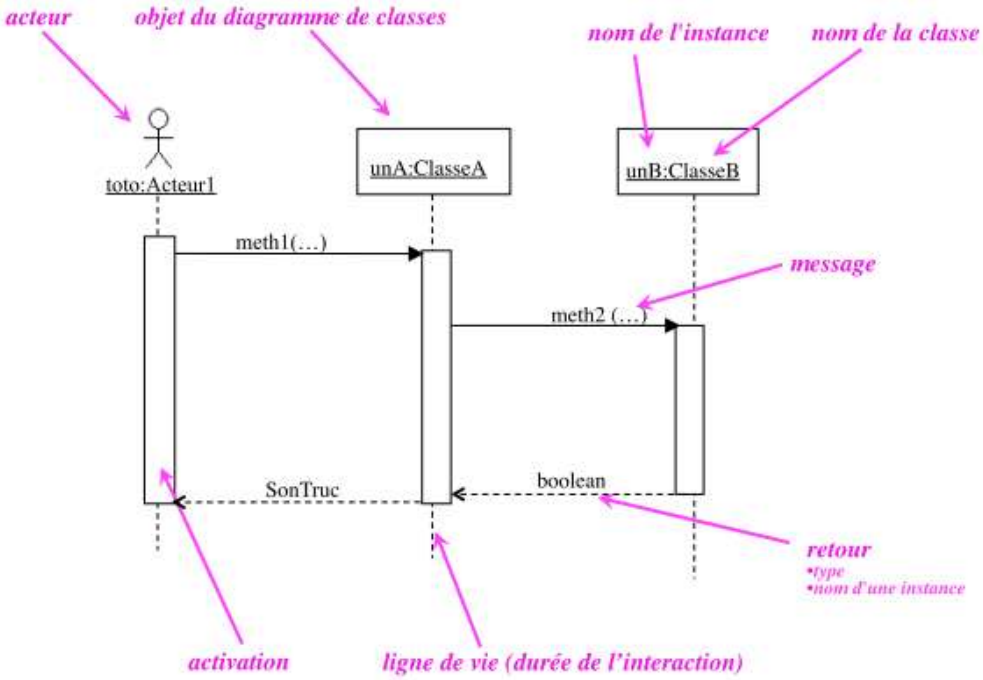


DIAGRAMME DE SÉQUENCE

Accent mis sur la communication, non sur la structure spatiale

- chaque objet est représenté par une barre verticale (sa ligne de vie)
- le temps s'écoule de haut en bas (numérotation des messages est optionnelle)



CONCLUSION UML

- Langage de modélisation objet
- Une notation, pas une méthode ou un processus
- Convient pour toutes les méthodes objet
- Dans le domaine public

AVANTAGES UML

- Concensus autour de son utilisation : standard dans l'industrie
- Notation avec une syntaxe très riche, tout en restant intuitive
- Intégration dans des ateliers de génie logiciel avec production de squelettes de codes et autres transformations automatiques des modèles

INCONVÉNIENTS UML

- Notation majoritairement graphique pouvant se révéler insuffisante ou trop chargée
- Sémantique floue ou mal définie pour certains types de diagrammes
- Lien parfois difficile entre les vues et diagrammes d'une même application

"Drawing UML diagrams is a reflection of making decisions about the object design.

The object design skills are what really matter, rather than knowing how to draw UML diagrams."

Applying UML and Patterns, Craig Larman

RÉFÉRENCES UTILISÉES

Ressources en ligne et figures

- C. Solnon, *INSA de Lyon - 3IF*
- E. Cariou, *Univ. Pau*
- A. Braffort, F. Terrier & D. Roussel

PRINCIPES SOLID

PROBLÈMES DE DÉVELOPPEMENT CLASSIQUES

Rigidité

Chaque changement cause une cascade de modifications dans les modules dépendants => une intervention a priori simple devient un cauchemar

Fragilité

Une modification du code a des répercussions sur des modules n'ayant aucune relation avec le code modifié

Immobilisme

Des modules ne peuvent pas être réutilisés par d'autres projets ou même par d'autres parties du même logiciel

Viscosité

Il est plus facile de faire un contournement plutôt que de respecter la conception qui avait été pensée

Opacité

Le code est difficile à lire et à comprendre

SOLID

Principes de Conception Orientée Objet proposés par Robert C. Martin (2002)

1. **S**ingle Responsibility Principle (SRP)
2. **O**pen-Closed Principle (OCP)
3. **L**iskov Substitution Principle (LSP)
4. **I**nterface Segregation Principle (ISP)
5. **D**ependency Inversion Principle (DIP)

“Common-sense solutions to common problems” : doivent être appliqués avec jugement, et surtout pratiqués

SINGLE RESPONSIBILITY

A class should have only one reason to change

Une classe ne devrait avoir qu'une seule responsabilité (un seul objectif fonctionnel)

On ne modifie la classe que si une unique spécification du logiciel change

Permet d'éviter les dépendances entre les modules (causes de rigidité et de fragilité)

OPEN-CLOSED

Software entities should be open for extension, but closed for modification

Une classe, une méthode, un module doivent pouvoir être étendus, supporter différentes implémentations (open) sans pour autant devoir être modifiés (closed)

Par exemple :

- Une classe ne peut être modifiée que pour corriger des erreurs. L'ajout de nouvelles fonctionnalités ne peut se faire que par la création de nouvelles classes (ex. des sous-classes de la première)
- On utilise au maximum le polymorphisme : des interfaces figées qui peuvent être implantées librement et augmentées par héritage

LISKOV SUBSTITUTION

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program

Les sous-classes doivent pouvoir remplacer leur classe de base

Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser “inconsciemment” des objets dérivés de cette classe

INTERFACE SEGREGATION

Many client-specific interfaces are better than one general-purpose interface

Il vaut mieux plusieurs interfaces spécifiques qu'une seule interface générale

Les classes clientes ne doivent pas être forcées de dépendre d'interfaces qu'elles n'utilisent pas

=> éviter les interfaces qui contiennent beaucoup de méthodes : quand elles grossissent, se poser la question de leurs rôles

DEPENDENCY INVERSION

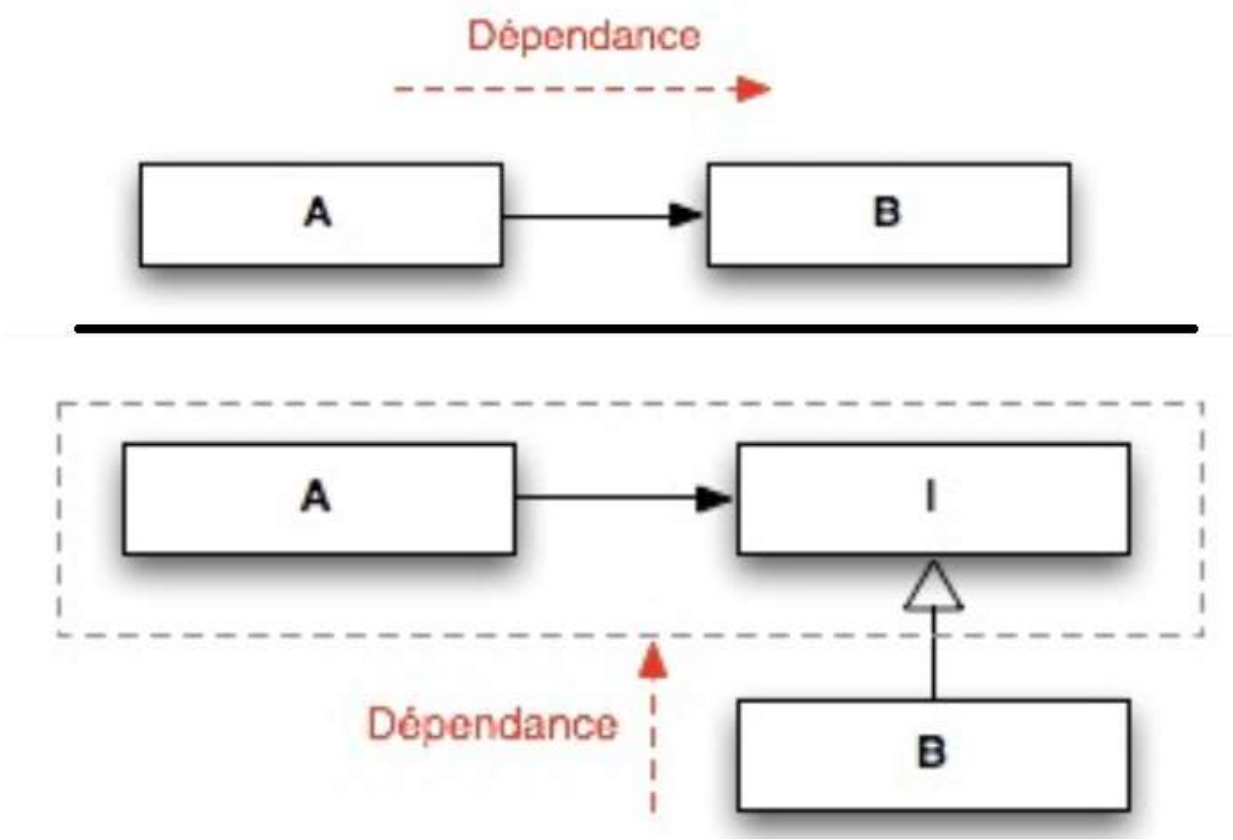
One should depend upon abstractions, not concretions

Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau.

Tous doivent dépendre d'abstractions

Ex : si A est “client” des services de B, A n'a pas à être modifié lorsque B est modifié, voire lorsque l'on veut une nouvelle implémentation C

DEPENDENCY INVERSION



- Le module haut niveau dépend d'une interface
- Le module bas niveau doit implémenter l'interface

DEPENDENCY INVERSION : EN PRATIQUE

Utiliser une interface ne suffit pas à découpler totalement A et B :

```
public class A{
  private I i;
  public A(){
    this.i = new B();
  }
}
```

Solution possible : **Injection de** la **dépendance** à la construction :

```
public class A{
  private I i;
  public A(I i){
    this.i = i;
  }
}
...
public static void main(String[] args)
      A a = new A(new B());
      ...
   }
```

PRINCIPES SOLID : EXEMPLES

CARRÉ ET RECTANGLE, CERCLE ET ELLIPSE

LISKOV SUBSTITUTION PRINCIPLE

Se poser la bonne question : pour les utilisateurs, le carré a-t-il le même comportement que le rectangle ?

Sinon, passer par une classe abstraite `RectangularShape`, spécialisée par `Square` d'un côté, et `Rectangle` de l'autre

CLASSER DES ÉTUDIANTS

SINGLE RESPONSABILITY PRINCIPLE

Si `Student` implements `Comparable`

- `Student` ne sait pas comment s'ordonner, puisque cela dépend du client
- Chaque fois que l'on a besoin d'un ordre différent, il faut modifier et recompiler

Solution : `Comparator`

- `StudentByName implements Comparator<Student>`
- `StudentBySection implements Comparator<Student>`

INSTANCIATIONS CONDITIONNELLES DANS UN CONSTRUCTEUR

```
public Car(EngineTypeEnum engineType) {
    if (engineType == EngineTypeEnum.FUEL) {
        Engine = new FuelEngine() ;
    } else if (...) {
        ...
    }
}
```

Problème : une nouvelle implémentation aura pour impact l'ajout d'une condition dans la méthode

INSTANCIATIONS CONDITIONNELLES

OPEN-CLOSED / DEPENDENCY INVERSION PRINCIPLES

Solution 1 : utiliser l'injection de dépendance

```
public Car (Engine engine) {
   this.engine = engine ;
}
```

Solution 2 : utiliser une fabrique d'objets dédiée (cf pattern *Factory*)

```
public Car (EngineType engineType) {
   engine = EngineFactory.getEngine(engineType) ;
}
```

AUTRES PRINCIPES

SOLID VS YAGNI

“You Ain't Gonna Need It”

Tolérance au changement vs productivité ?

Pas obligatoire de rendre toutes les modules abstraits pour réduire le couplage, ou toutes les créations d'objet indépendantes des implémentations sous peine d'avoir plus d'interfaces et de fabriques que de classes concrètes

Possible de commencer l'implémentation sans suivre l'ensemble des principes SOLID, et de les mettre en œuvre quand nécessaire (ajout d'un cas particulier, d'une condition, d'une nouvelle implémentation...)

L'apparition d'instanciations conditionnelles, de conditions sur le type d'une classe... sont des signaux en faveur d'un éventuel refactoring

DRY VS WET

“Don't Repeat Yourself”

Chaque connaissance dans un système doit avoir une représentation unique, sans ambiguïté et fiable

Si principe bien appliqué, on évite les problèmes de rigidité et fragilité, les modifications sont uniformes et prévisibles

WET = “Write Everything Twice”, “We Enjoy Typing”, “Waste Everyone's Time”

KISS

“Keep It Simple, Stupid”

“There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult”

C. A. R. Hoare, The Emperor's Old Clothes