UML

Le langage de modélisation unifié

Stéphane Lopes
stephane.lopes@prism.uvsq.fr

2013-2014

Chapitre I : Préambule

Objectifs du cours

- Présenter la notation UML
- Suivre une progression proche d'un processus de développement
- Présenter le langage OCL
- Aborder la méta-modélisation

Prérequis

- Connaissance des concepts objets

Plan général

- Introduction
- Exprimer les besoins
- Modéliser le domaine
- Spécifier l'architecture de l'application
- Conception OO
- Architecture de déploiement
- Langage OCL
- Métamodélisation

Références

Bibliographie

- *UML 2 et les design patterns*, **Craig Larman**, Pearson, 2005 (UVSQ : 005.12 LAR)
- *UML 2.0*, **Martin Fowler**, CampusPress, 2004 (UVSQ : 005.12 FOW)
- *UML 2 - de l'apprentissage à la pratique*, **Laurent Audibert**, Ellipses, 2009 (UVSQ : 005.12 AUD)
- *UML 2 par la pratique* , **Pascal Roques**, Eyrolles, 2009 (UVSQ : 005.12 ROQ)
- *MDA en action*, **Xavier Blanc**, Eyrolles, 2005 (UVSQ : 005.12 BLA)

Références

Webographie

- Ressources sur UML sur le site de l'OMG
  - UML Infrastructure 2.4.1 (août 2011)
  - UML Superstructure 2.4.1 (août 2011)
- UML sur Wikipedia (fr, en)
- Le site *UML en français*
- Cours UML 2 de Laurent Audibert sur Developpez.com
- Introduction to the Diagrams of UML 2.0, Scott W. Ambler, Agile Modeling
- Site UML-Diagrams
- Best Practices for Applying UML

Chapitre II : Introduction

Chapitre II : Introduction

Chapitre II : Introduction

4. Qu'est-ce qu'UML ?
   - Définition et contenu
   - Diagrammes
   - Autres standards

UML

Definition

UML (*Unified Modeling Language*) est un ensemble de notations graphiques s'appuyant sur un métamodèle et permettant de spécifier, visualiser et documenter des systèmes logiciels orientés-objet.

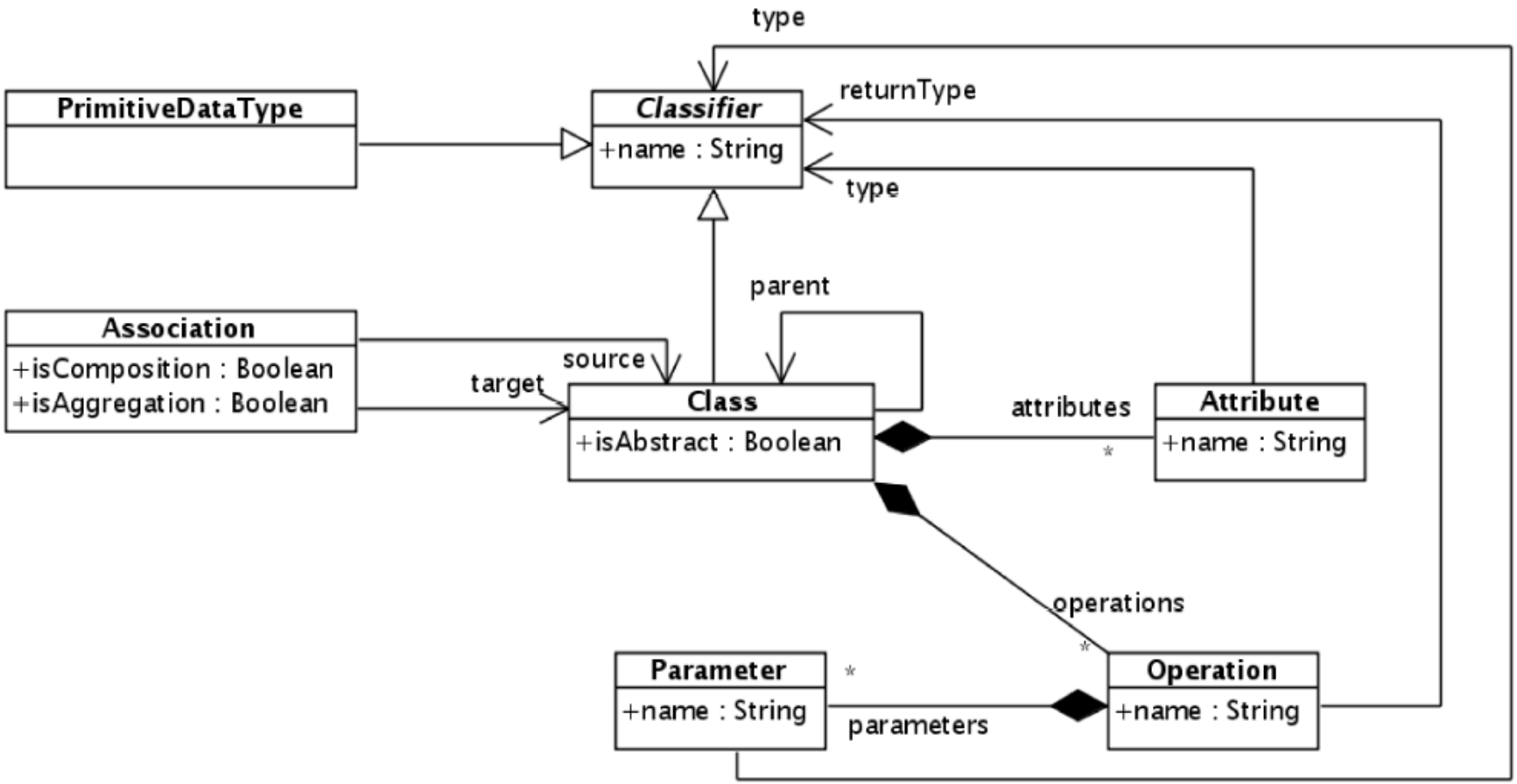
- UML est un standard contrôlé par l'OMG (*Object Management Group*)
- UML est le fruit de l'unification de plusieurs langages graphiques de modélisation objet

Contenu

- UML définit une *notation* et un *métamodèle*
- La notation est la représentation graphique des modèles
- Elle représente la syntaxe du langage de modélisation
- Le métamodèle apporte des éléments complémentaires pour préciser la signification des notations, i.e. leur *sémantique*
- Le métamodèle d'UML est représenté par des diagrammes de classes et des descriptions textuelles

Exemple

Un extrait du métamodèle d'UML



source : Sample UML Metamodel (eclipse)

Ce que n'est pas UML

- UML n'est pas l'A/COO
  - connaître UML n'implique pas de savoir analyser et concevoir un système OO
- UML n'est pas une méthode
  - la précification UML ne donne pas d'indice sur la façon d'utiliser les diagrammes
- UML n'est pas une approche formelle
  - les éléments du langage ne sont pas définis formellement

UML ne suffit pas

- Tous les diagrammes UML ne sont pas utiles dans chaque application
- Il existe d'autres documents utiles pour la description d'une application
  - glossaires, enchaînement des écrans, ...
- Il ne faut pas confondre un modèle et le diagramme le représentant
  - on représente généralement un *cas d'utilisation* par un document textuel

Chapitre II : Introduction

4. Qu'est-ce qu'UML ?
   - Définition et contenu
   - Diagrammes
   - Autres standards

Diagrammes UML

- UML définit deux grandes familles de diagrammes
  - les *diagrammes de structure* spécifient les aspects statiques d'un système
  - les *diagrammes de comportement* s'intéressent aux aspects dynamiques
  - dans cette catégorie, les *diagrammes d'interaction* décrivent les échanges entre objets
- UML 2.4.1 (août 2011) définit 14 diagrammes

Liste des diagrammes I

Activités	processus métier ou logique d'une partie du système
Cas d'utilisation	relations entre les acteurs et les cas d'utilisation
Classes	éléments statiques du modèle et leurs relations
Communication	intéractions entre instances (organisation structurelle)
Composants	composants, interfaces et relations entre les éléments du système
Déploiement	architecture d'exécution du système
Etats-transitions	états et changements d'état d'un objet
Objets	instantané de la structure des objets
Paquetages	organisation des éléments de modélisation
Profil	extension d'UML
Séquence	intéractions entre instances (ordre des messages dans le temps)

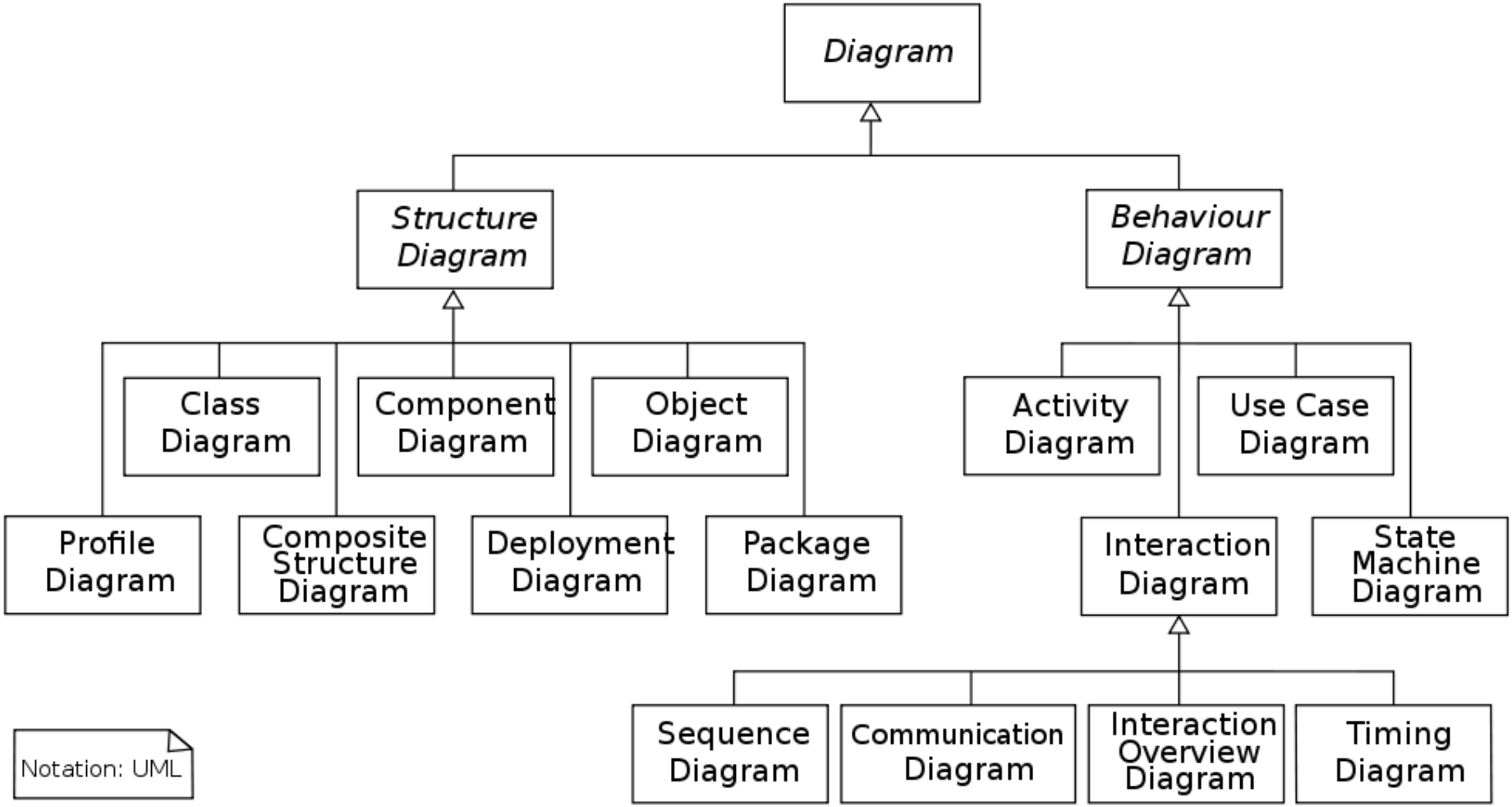
Liste des diagrammes II

Structure composite structure interne d'un classeur avec ses points d'intéraction

Temps changements d'états d'un classeur dans le temps

Vue d'ensemble des interactions flôt de contrôle d'un système (à grande échelle)

Structuration des diagrammes



source : UML diagrams overview (Wikipedia)

UML et les autres standards

BPMN *Business Process Modeling Notation* est une notation pour représenter l'orchestration des processus métier d'une organisation

EMF *Eclipse Modeling Framework* est une framework de modélisation de la fondation *Eclipse* proche de MDA

MDA *Model Driven Architecture* est une approche du développement logiciel s'appuyant sur les modèles et leurs transformations

MOF *Meta-Object Facility* est un métamétamodèle proposé par l'OMG

SYSML *Systems Modeling Language* est un profil UMI pour la description de systèmes

XMI *XML Metadata Interchange* permet l'échange de modèles MOF

Chapitre II : Introduction

Différents modes d'utilisation

UML peut être utilisé en pro-ingénierie ou en rétro-ingénierie selon trois modes :

- **Esquisse** support de discussion
- **Plan** spécification pour l'équipe de programmeurs
- **Spécification exécutable** spécification de la totalité du système

UML en mode esquisse

- Les diagrammes servent de base à la discussion
- Uniquement les aspects les plus importants
- Utilise un sous-ensemble de la syntaxe
- La syntaxe est parfois approximative
- Fait « à la main » ou avec des outils simples

UML en mode plan

- Représentation exhaustive de la conception
- Sert de référence pour les programmeurs
- Nécessite des outils évolués

UML pour des spécifications exécutables

- Permet d'automatiser la tâche de programmation
- UML est alors vu comme un langage de programmation
- Nécessite des outils sophistiqués
- Approche MDA (*Model Driven Architecture*) et MDE (*Model Driven Engineering*)

Chapitre II : Introduction

Avant UML

- Début des années 90
- Coexistence de plusieurs **méthodes** d'A/COO
- Des concepts similaires sont représentés par des notations variées

Avant UML

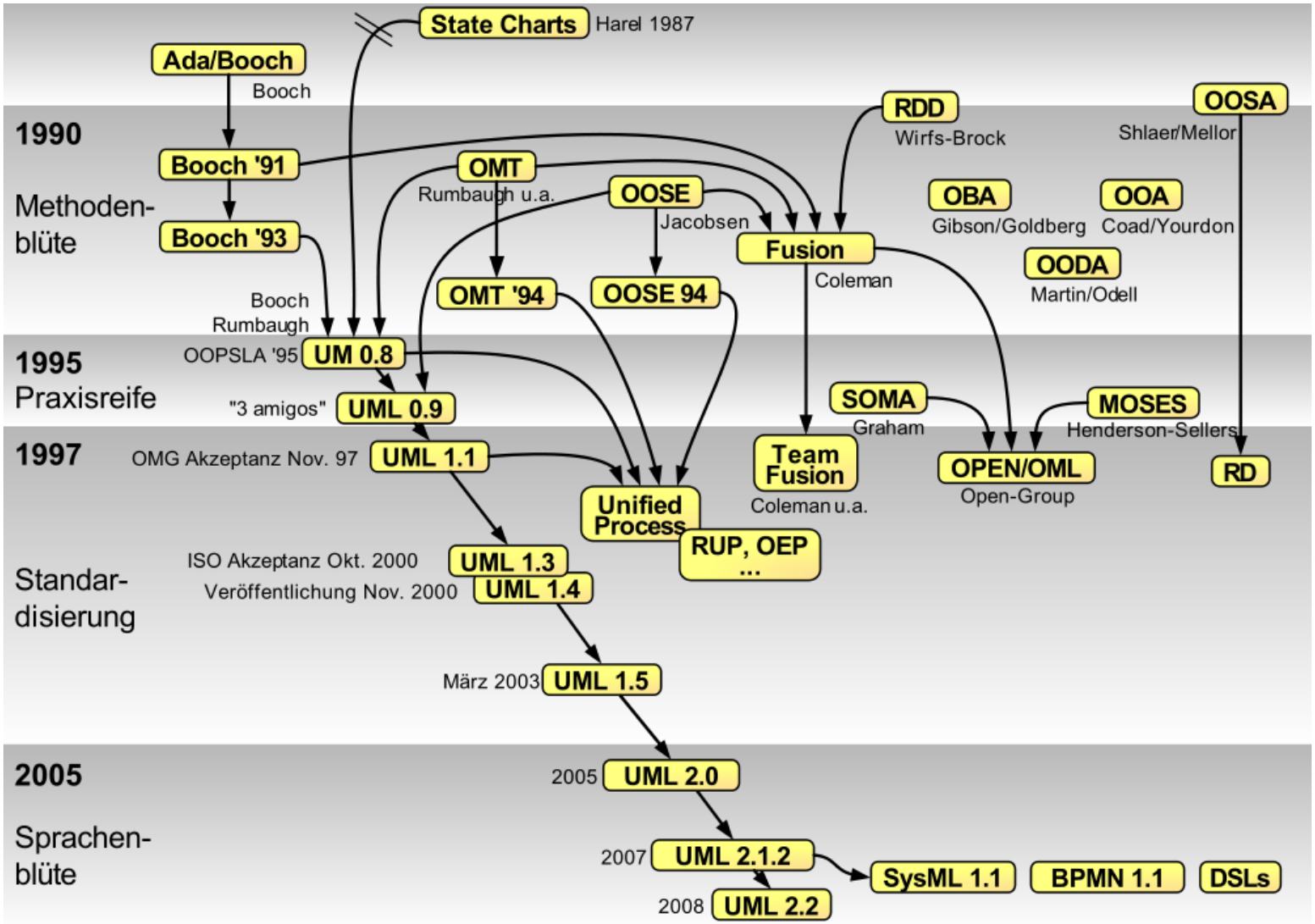
Quelques méthodes

- OOAD, Grady Booch
  - *Object-oriented analysis and design (OOAD)*, **Grady Booch**, Addison-Wesley, 1994
- OOA/OOD, Peter Coad et Edward Yourdon
  - *Object-Oriented Analysis*, **Peter Coad, Edward Yourdon**, Yourdon Press, 1991
  - *Object-Oriented Design*, **Peter Coad, Edward Yourdon**, Yourdon Press, 1991
- OOSE, Ivar Jacobson pour Objectory
  - *Object-Oriented Software Engineering : A Use Case Driven Approach*, **Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard**, Addison-Wesley, 1992
- OMT, Jim Rumbaugh
  - *Object-Oriented Modeling and Design*, **James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen**, Prentice Hall, 1991

Les prémisses d'UML

- En 1994, Rumbaugh rejoint Booch chez Rational
- En 1995, version 0.8 de la « méthode unifiée », rachat d'Objectory par Rational, Jacobson se joint à l'équipe
- En 1996, l'OMG s'implique dans le développement de la méthode
- En 1997, Rational publie la version 1.0 d'UML et la soumet à l'OMG

Historique des méthodes d'A/COO



source : History of object-oriented methods and notations (Wikipedia)

Chapitre II : Introduction

Chapitre II : Introduction

7. Mise en œuvre d'UML
   - Cycles de développement
   - Processus unifié
   - UML et processus unifié

Cycle de vie d'une application

- La gestion du cycle de vie d'une application (*Application lifecycle management* ou *ALM*) est un processus continu de suivi d'une application
- L'ALM regroupe la gestion de processus métier et les processus de développement logiciel
- Organise les activités de développement tout au long de la vie d'une application

Principales activités

- Modélisation métier
- Gestion des exigences
- Analyse et conception
- Implémentation
- Tests
- Déploiement
- Gestion des configurations
- Gestion de projet

Cycle linéaire

- Les activités sont enchaînées avec peu de possibilités de retour en arrière
- Chaque activité est censée être terminée avant qu'une autre ne démarre
- Cycle en cascade (années 1970)
  - les livrables d'une phase servent de point de départ pour la suivante
- Cycle en V (années 1980)
  - à chaque phase correspond une phase de validation

Cycle itératif et incrémental

- Le processus global est un ensemble d'itérations
- Chaque itération conduit à une version exécutable de l'application qui peut être testée
- Une itération peut introduire de nouveaux besoins
- Cycle en spirale
  - 1 itération : définir les objectifs, identifier les risques, développer et tester, plannifier l'itération suivante
- Processus unifié
  - méthode pilotée par les cas d'utilisation, centrée sur l'architecture, itérative et incrémentale
- Méthode agile
  - Manifesto for Agile Software Development
  - valeurs fondamentales : les individus et leurs interactions, des logiciels opérationnels, la collaboration avec les clients, l'adaptation au changement
  - quelques méthodes : Scrum, eXtreme Programming

Chapitre II : Introduction

7. Mise en œuvre d'UML
   - Cycles de développement
   - Processus unifié
   - UML et processus unifié

Introduction

- UP (*Unified Process*) est né de la fusion des travaux de Booch, Rumbaugh et Jacobson
- Lr processus unifié fournit un cadre général qu'il faut instancier
- Le *Rational UP* en est la version commerciale la plus répandue
- D'autres déclinaisons existent : Agile UP, Open UP

Caractéristiques

Piloté par les cas d'utilisation la réalisation du logiciel s'appuie sur les besoins fonctionnels

Centré sur l'architecture les choix d'architecture sont cruciaux pour la réussite du projet

itératif et incrémental la priorité est donnée aux cas d'utilisation les plus risqués

Phases

Inception (initialisation) définition des objectifs et plannification

Élaboration stabilisation de l'architecture, identification et élimination des risques majeures

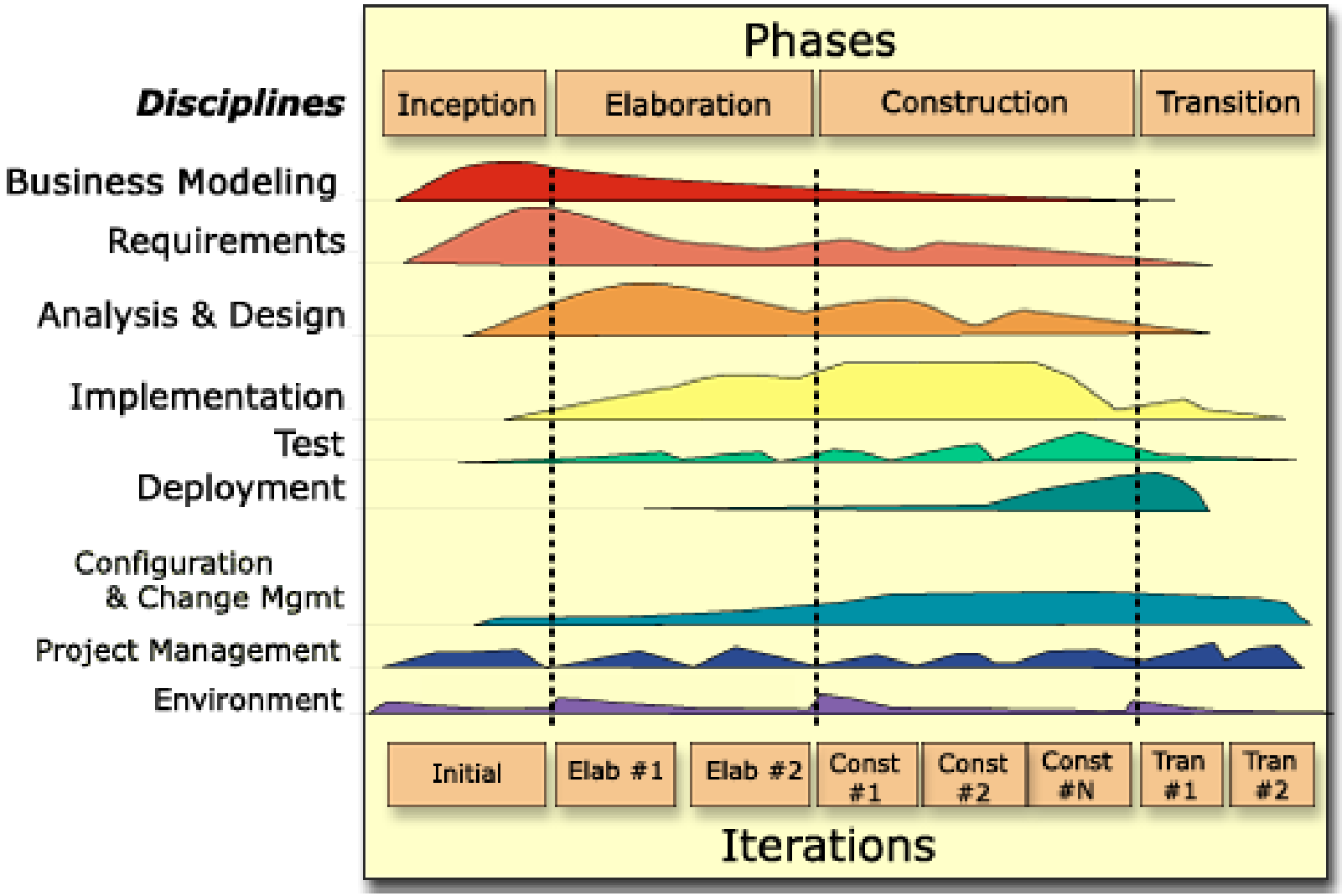
Construction conception et réalisation des cas d'utilisation

Transition finalisation et livraison

Disciplines

- Une *discipline* est un ensemble d'*activité* qui produisent des *artefacts*
- Disciplines d'ingénierie (*workflows du processus*)
  - Modélisation métier
  - Gestion des exigences
  - Analyse et conception
  - Implémentation
  - Tests
  - Déploiement
- Disciplines de support (*workflows de soutien*)
  - Gestion de la configuration et des changements
  - Gestion de projet
  - Environnement

Phases, disciplines et itérations



source : Rational Unified Process

Chapitre II : Introduction

7. Mise en œuvre d'UML
   - Cycles de développement
   - Processus unifié
   - UML et processus unifié

Mise en œuvre d'UML dans UP

- Les disciplines d'UP produisent des artefacts
- Certains de ces artefacts peuvent être représentés partiellement ou complétement par des diagrammes UML
- Un diagramme peut être commencé durant une itération et complété par la suite
- Les artefacts et donc les diagrammes utilisés dépendent de la variante d'UP choisie

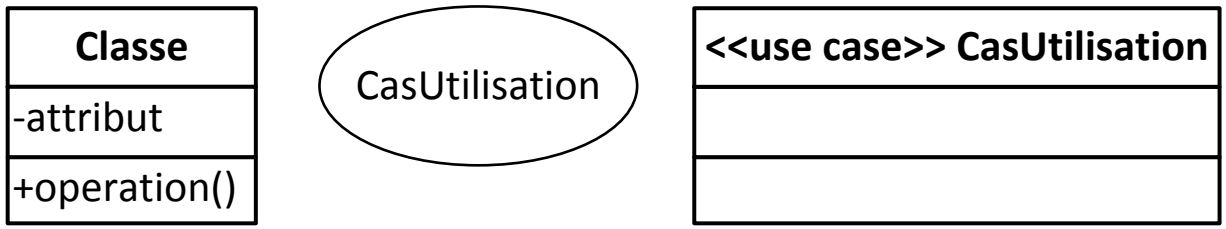
Chapitre II : Introduction

Espace de noms

- Un espace de noms (*namespace*) est un conteneur pour des éléments nommés
- Un espace de noms peut être contenu dans un autre espace de noms
- Un élément nommé peut être référencé par son nom qualifié (*namespace-name*::*element-name*)
- Un espace de noms ne possède pas de représentation graphique

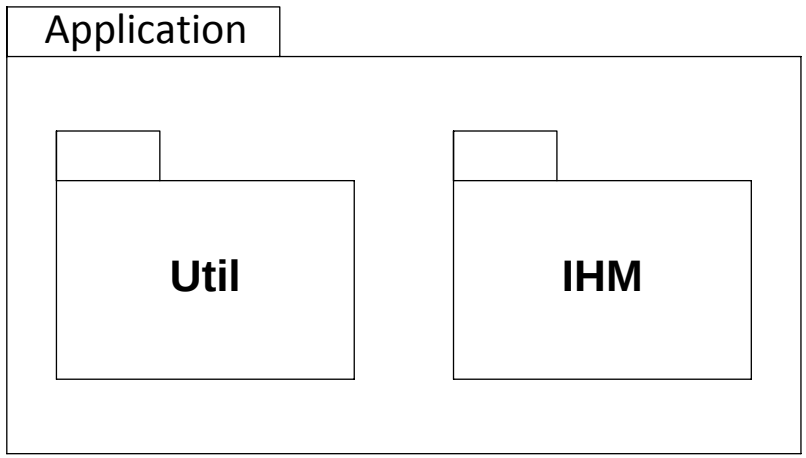
Classeur

- Un classeur (*classifier*) permet de décrire un ensemble d'instances possédant des caractéristiques communes (propriétés ou opérations)
- Un classeur est un espace de nom
- Il est possible de définir des relations de généralisation entre classeurs
- Un classeur est représenté graphiquement par un rectangle contenant le nom du classeur et d'éventuels compartiments
- Le type de classeur peut apparaître entre guillemets avant son nom



Paquetage

- Un paquetage (*package*) est un espace de nom regroupant des éléments
- Les paquetages peuvent être imbriqués
- Il est représenté comme une pochette à onglet
- Le nom du paquetage apparaît sur la pochette ou dans l'onglet

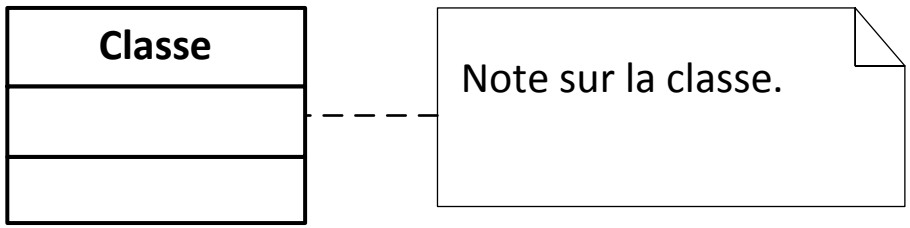


Mot-clé, stéréotype et tagged value

- Un mot-clé (*keyword*) est définit dans le métamodèle UML et apparaît entre guillemets ou entre accolades
- Un stéréotype (*stereotype*) fait partie des mécanismes d'extension d'UML
- Il précise comment une métaclasse existante est étendue par un profil
- Il est appliqué à un élément de modèle en le faisant apparaître entre guillemets devant le nom de l'élément
- Il peut posséder des propriétés
- Ces propriétés apparaissent comme des couples (attribut, valeur) (*tagged value*) au niveau de l'instance du stéréotype

Note

- Un commentaire (*note*) est un élément représentant une annotation textuelle
- Une note peut être attachée à plusieurs éléments
- Une note est représentée par un rectangle corné et reliée aux éléments annotés par une ligne pointillée



Chapitre III : Exprimer les besoins

Chapitre III : Exprimer les besoins

Définition

- Les *besoins* sont les conditions qu'un projet doit satisfaire
- Dans une approche itérative, les besoins évoluent durant le projet
- La collecte des besoins est une tâche cruciale faisant appel à différentes techniques (rédaction des cas d'utilisation avec le client, ateliers d'expressions des besoins, . . .)

Classification des besoins

L'exemple du modèle FURPS+

Functionality	Fonctionnalité, capacités et sécurité
Usability	Facteurs humains, aide et documentation
Reliability	Fiabilité, fréquence des pannes, récupération
Performance	Temps de réponse, disponibilité
Supportability	Adaptabilité, internationalisation, facilité de maintenance
Implémentation	Langages et outils, matériel
Interface	Interactions avec des systèmes externes
Exploitation	Gestion du système dans l'environnement de production
Conditionnement	
Aspects juridiques	Licences

Intérêt

Ces classifications ont pour objectif d'éviter de négliger un aspect du système.

Expressions des besoins dans UP

Les artefacts

Modèle de cas d'utilisation décrit les besoins fonctionnels

Spécifications supplémentaires exprime les besoins non fonctionnels

Glossaire représente le dictionnaire de données

Vision donne un aperçu de haut niveau du projet

Règles métier décrivent les règles du domaine non spécifiques à une application

Analyse des besoins et UP

Phases d'inception et d'élaboration

- La phase d'inception n'est pas la phase de définition des besoins
- C'est une courte phase initiale qui permet de déterminer l'opportunité du projet
- Elle comprend l'analyse
  - d'une petite partie des besoins fonctionnelles,
  - des besoins non fonctionnels critiques.
- Les besoins sont ensuite enrichis durant la phase d'élaboration voire de construction

Chapitre III : Exprimer les besoins

Chapitre III : Exprimer les besoins

Qu'est qu'un cas d'utilisation ?

- Un cas d'utilisation est une description textuelle de la façon dont des *acteurs* vont utiliser le système pour atteindre un *but*
- Les cas d'utilisation ne sont pas des diagrammes mais bien des textes

Remarque

Les cas d'utilisation ne sont pas *orientés objet*

Concepts

Sujet système en cours d'étude

Acteur entité qui possède un comportement (personne ou système)

- **principal** possède des buts qu'il atteint en utilisant le système
- **auxiliaire** fournit un service au système
- **hors-champs** est concerné mais n'est pas principal ou auxiliaire

Scénario séquence d'actions et d'interactions entre un ou plusieurs acteurs et le système. On parle aussi *d'instance de cas d'utilisation*.

Cas d'utilisation collection de scénarios décrivant la façon dont un ou plusieurs acteurs va utiliser le système pour atteindre un *but*

Niveaux de détails

abrégé	résumé succinct du scénario de base en un paragraphe
informel	un paragraphe par scénario
détaillé	les étapes et les variantes sont détaillées

Sections d'un cas d'utilisation I

Nom Nom du cas d'utilisation (commence par un verbe).

Périmètre Le système en cours de conception (*cas d'utilisation système*). Un cas d'utilisation peut aussi décrire les interactions au sein d'un domaine (*cas d'utilisation métier*).

Niveau But utilisateur ou sous-fonction (factorise les étapes communes à plusieurs cas d'utilisation).

Acteur principal Atteint son but grâce au système.

Parties prenantes et intérêts Qui est concerné par ce cas d'utilisation ? Ce que ça leur apporte.

Préconditions Conditions initiales à remplir pour que le cas d'utilisation se déroule.

Garanties en cas de succès Ce qui est réalisé (postconditions).

Sections d'un cas d'utilisation II

Scénario principal Scénario de succès (« *happy path* ») sans conditions ni branchements.

Extensions Scénarios alternatifs de succès ou d'échec. Ce sont des branches qui partent du scénario principal.

Spécifications particulières Besoins non fonctionnels directement liés au cas d'utilisation.

Liste de variantes des données et des technologies Méthdes d'E/S et formats de données.

Fréquence d'occurence « Importance » du cas d'utilisation.

Divers Questions ouvertes.

Recommandation pour la rédaction de cas d'utilisation

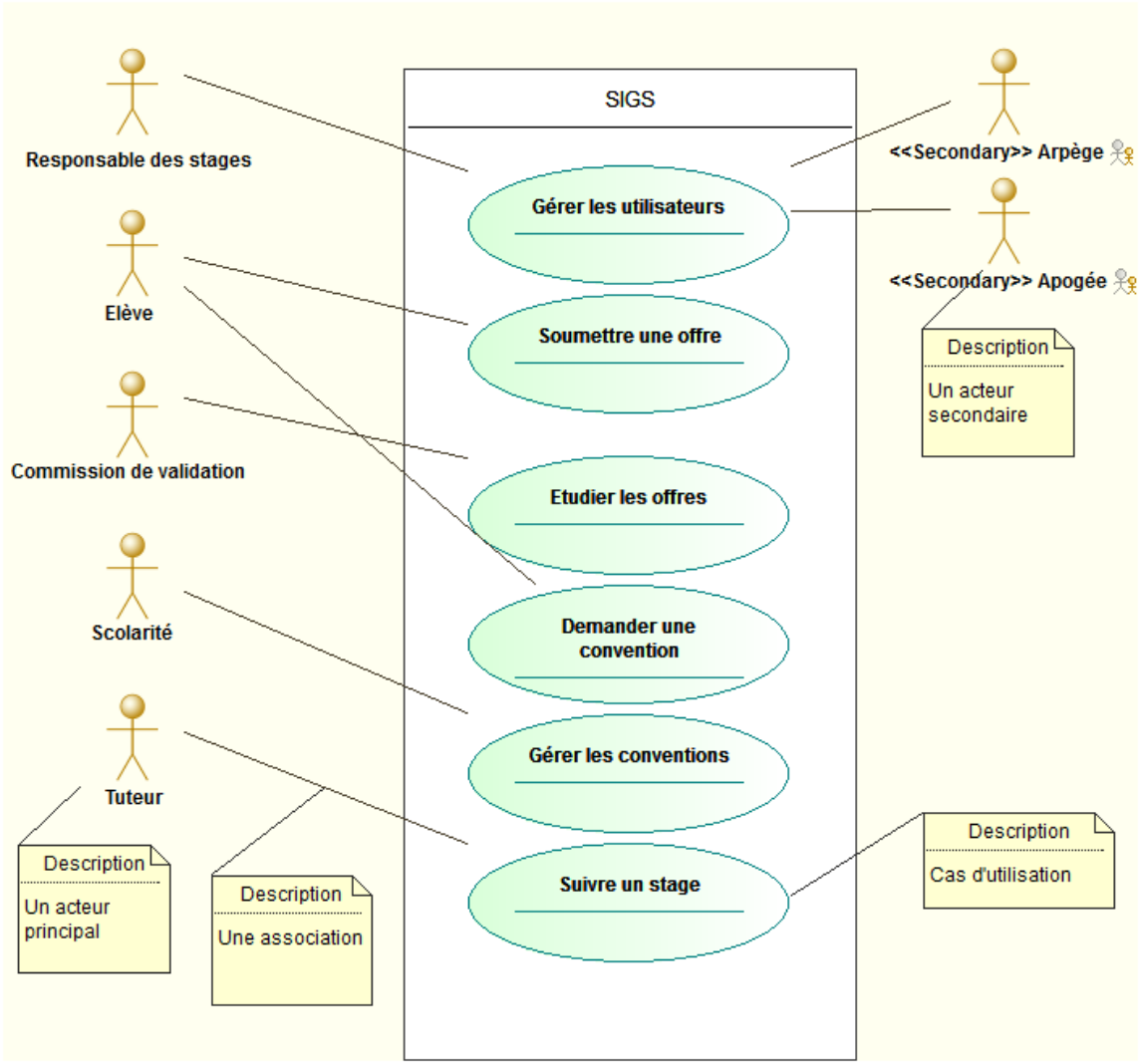
- Utiliser le style « essentiel » : se concentrer sur l'intention de l'acteur et non sur l'IHM.
- Rédiger avec concision.
- Rédiger des cas d'utilisation en « boîte noire » : ne pas décrire le fonctionnement interne du système (le comment) mais les *responsabilités* (le quoi).
- Doit être centré sur les acteurs et leurs buts : les résultats doivent avoir de la valeur pour les acteurs.

Chapitre III : Exprimer les besoins

Diagramme de cas d'utilisation

- Le *diagramme de cas d'utilisation* d'UML permet de représenter :
  - les noms de cas d'utilisation,
  - les acteurs,
  - les relations entre acteur et cas d'utilisation.
- Il fait partie des diagrammes de comportement d'UML

Exemple de diagramme de cas d'utilisation



Extrait d'un diagramme de cas d'utilisation pour l'étude de cas stages

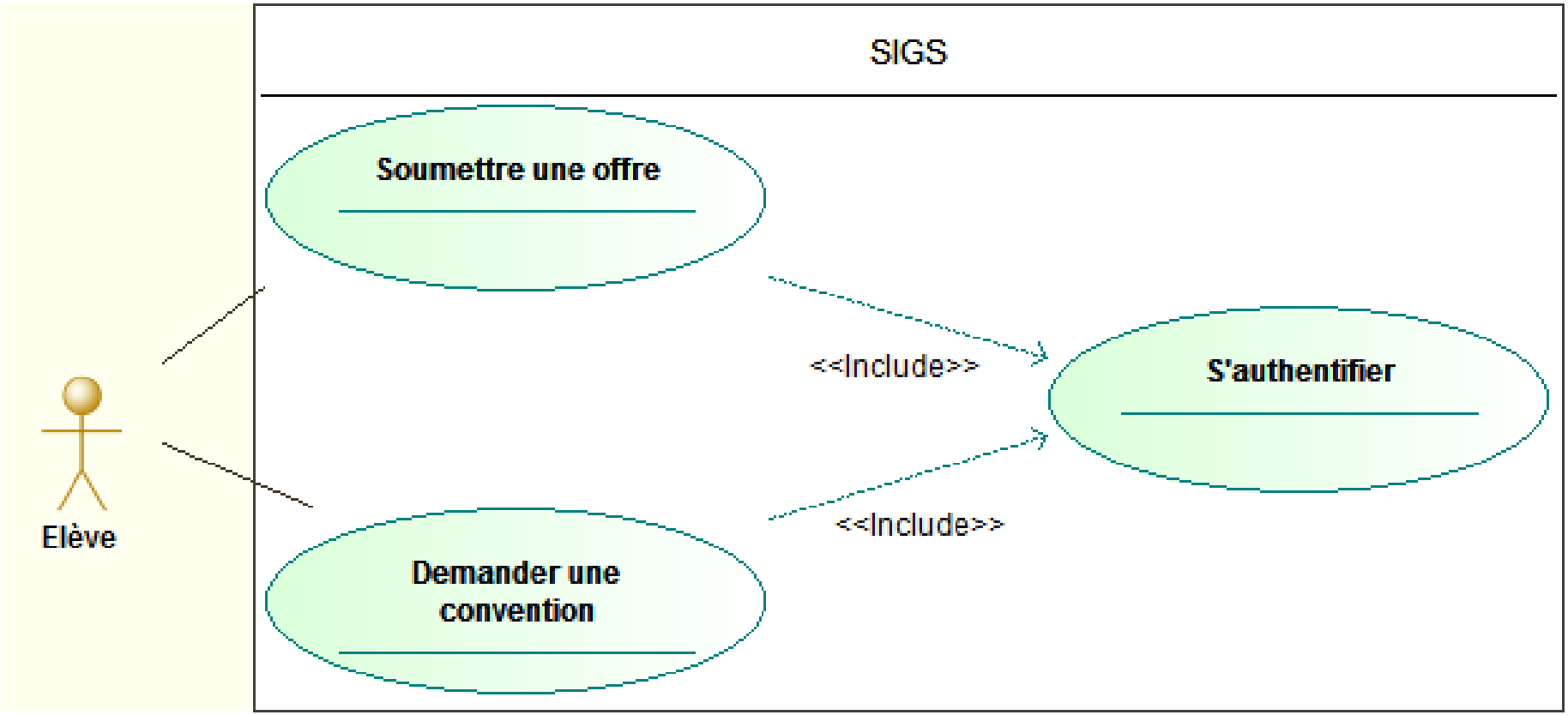
Notations du diagramme de cas d'utilisation

- Un cas d'utilisation est représenté par une ellipse contenant le nom du cas d'utilisation ou comme un classeur marqué par le stéréotype « `use case` »
- Un rectangle contenant les cas d'utilisation et le nom du sujet modélisé marque les limites du système
- Les acteurs se trouvent à l'extérieur de ces limites
- Un acteur est représenté par un symbole spécifique ou par un classeur marqué par le stéréotype « `actor` »
- Des relations d'association relient acteurs et cas d'utilisation

Relations entre cas d'utilisation

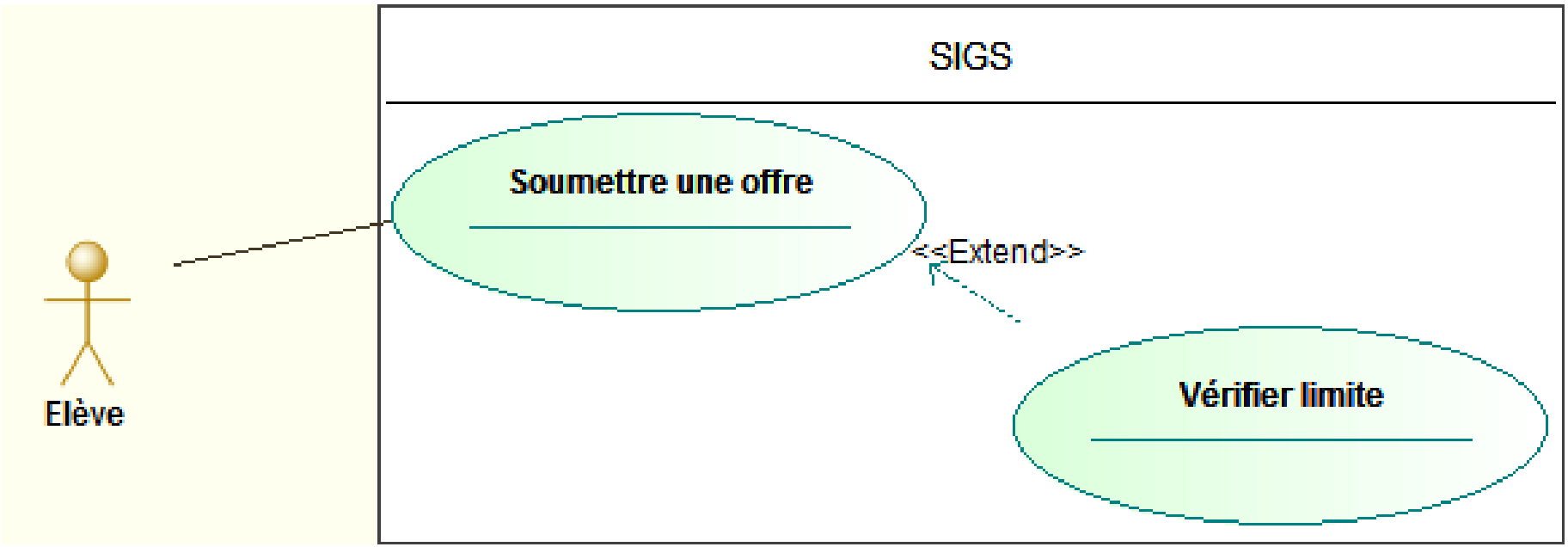
- Un cas d'utilisation A *inclut* un cas B si le comportement décrit par le cas A inclut le comportement du cas B
- Lorsque A est sollicité, B l'est obligatoirement aussi
- Cette relation est représentée par une dépendance (flêche pointillée) de A vers B marquée par le stéréotype « `include` »
- Un cas d'utilisation A *étend* un cas B si le cas A peut être appelé au cours de l'exécution de B
- Cette relation est représentée par une dépendance de A vers B marquée par le stéréotype « `extend` »
- Un cas d'utilisation A *est une généralisation* d'un cas B si B est un cas particulier de A
- Cette relation est représentée comme un lien d'héritage entre B et A

Exemple de relation d'inclusion



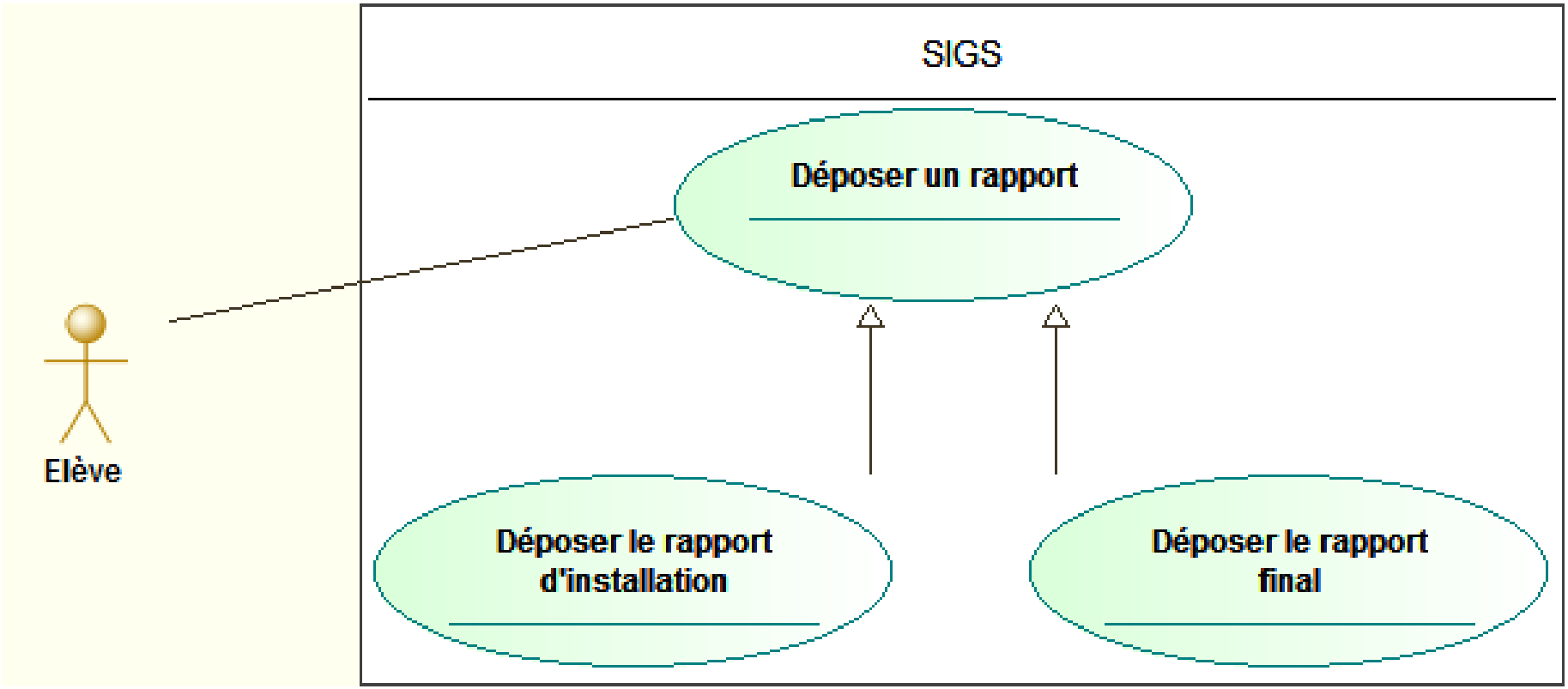
Extrait d'un diagramme de cas d'utilisation pour l'étude de cas stages

Exemple de relation d'extension



Extrait d'un diagramme de cas d'utilisation pour l'étude de cas stages

Exemple de généralisation entre cas d'utilisation

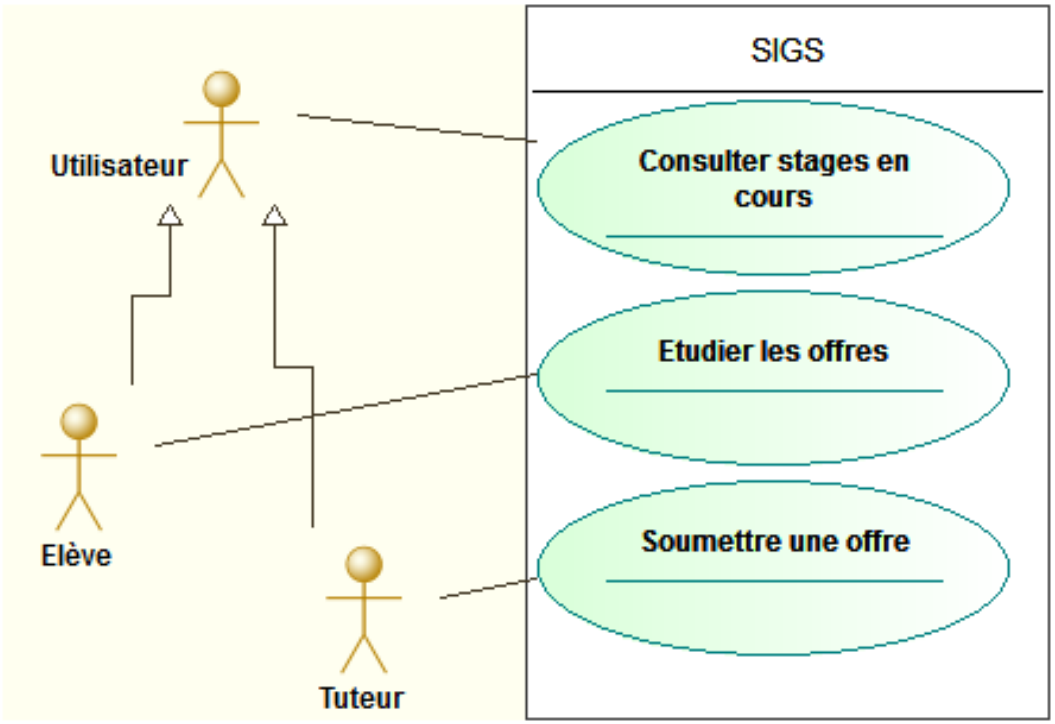


Extrait d'un diagramme de cas d'utilisation pour l'étude de cas stages

Relations entre acteurs

- Un acteur A *est une généralisation* d'un acteur B si A peut se substituer à B
- Cette relation est représentée comme un lien d'héritage entre B et A
- Tous les cas d'utilisation accessibles à A le sont aussi à B

Exemple de généralisation entre acteurs



Extrait d'un diagramme de cas d'utilisation pour l'étude de cas stages

Recommandations à propos du diagramme de cas d'utilisation

- Ne pas consacrer trop d'efforts aux diagrammes mais plutôt aux descriptions textuelles
- Ce diagramme sert de table des matières graphique pour les cas d'utilisation du système et permet de visualiser les limites du système
- Les relations d'inclusion, d'extension, . . . sont à utiliser avec parcimonie : elles rendent le diagramme plus complexe et il est préférable de se focaliser sur le texte.

Chapitre III : Exprimer les besoins

10 Besoins fonctionnels
- Cas d'utilisation
- Diagramme de cas d'utilisation
- Diagramme de séquence système

Diagramme de séquence système

- Un *diagramme de séquence système* (DSS) est un artefact illustrant les entrées/sorties du système
- Il fait parti du modèle de cas d'utilisation
- Le diagramme de séquence d'UML est bien adapté à la représentation de cet artefact

Intérêt des DSS

- Un DSS explicite un scénario de cas d'utilisation
- Il représente les évènements générés par les acteurs et leur ordre vis à vis de système
- Les systèmes sont considérés comme des boites noires
- Un DSS est créé pour chaque scénario complexe d'un cas d'utilisation

Introduction au diagramme de séquence UML

- Un diagramme de séquence fait parti des diagrammes d'interaction d'UML
- L'objectif des diagrammes d'interaction est de mettre en évidence le rôle des participants ainsi que les messages échangés
- Un diagramme de séquence permet de représenter l'ordre des échanges de messages entre des instances
- Dans un DSS, les instances représentent le système et les acteurs
- Lors de la conception, les instances sont les objets logiciels de l'application

Principaux concepts

Participant	élément impliqué dans l'interaction
Ligne de vie	représente un participant de l'interaction
Message	description d'une interaction
Évènement	envoie/réception d'un message, signal, ...

Exemple de DSS

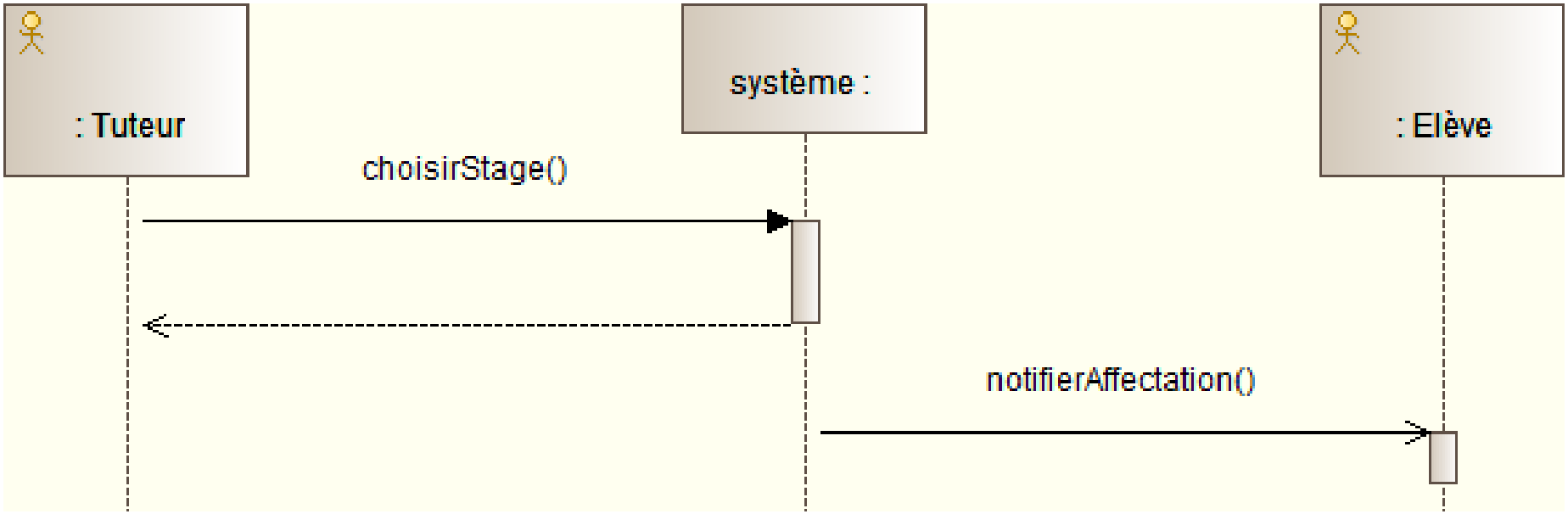


Diagramme de séquence « Tutorer un stage »

Notation

- Une ligne de vie est représentée par un rectangle accroché à une ligne verticale pointillée
- Le rectangle contient le nom du participant et/ou son type (séparé par « : »)
- Un message est représenté par une flèche en trait plein dirigée vers la cible
  - pointe ouverte : message asynchrone
  - pointe pleine : message synchrone
- Un message synchrone peut être suivi d'une réponse (flèche en pointillée)
- Un rectangle sur une ligne de vie marque l'exécution d'un traitement

Chapitre III : Exprimer les besoins

Besoins non fonctionnels

- Les besoins fonctionnels ne couvrent qu'une partie des besoins d'une application
- Les besoins restant sont nommés *besoins non fonctionnels*
- La classification FURPS en montre différents types (performance, . . .)
- Ces besoins ne sont pas décrits par le modèle de cas d'utilisation mais par d'autres documents

Expressions des besoins non fonctionnels dans UP

Les artefacts

Spécifications supplémentaires capturent les autres besoins comme le *packaging*, les licences, ...

Glossaire représente le dictionnaire de données

Vision communique les idées importantes du projet

Règles métier décrivent les règles du domaine non spécifiques à une application

Chapitre IV : Modéliser le domaine

Chapitre IV : Modéliser le domaine

Modélisation métier

- La *modélisation métier* est une discipline de UP
- Elle rend compte de la « structure et la dynamique de l'organisation où doit être déployé un système »
- Le *modèle objet métier* (*Business Objet Model* ou *BOM*) est un artefact de cette discipline
- Il représente à la fois les aspects statiques (modèle du domaine) et dynamiques (modèle des processus métier) d'un domaine

Modèle du domaine

- Le *modèle du domaine* est un modèle statique très important en A/COO
- Il permet de visualiser les concepts métier (classes conceptuelles) et leurs relations
- Il servira de source d'inspiration pour un certain nombre d'objets logiciels
- Cet artefact UP peut êre constitué d'un ensemble de diagrammes de classes UML (sans opération)

Ce que n'est pas le modèle du domaine

- Il ne représente pas des objets logiciels
- Ce n'est pas un ensemble de diagrammes décrivant les classes logicielles
- Ce n'est pas une représentation de la couche domaine d'une architecture logicielle
- Ce n'est pas un modèle de données

Classe conceptuelle

- Une *classe conceptuelle* est une entité du domaine étudié
- Elle comporte :
  - un *nom* qui identifie le concept,
  - une *intension* qui la définit en terme de propriétés,
  - une *extension* qui précise l'ensemble des instances de ce concept.

Modèle des processus métier

- Le *modèle des processus métier* (*Business Process Model* ou *BPM*) est un modèle dynamique
- Il permet de visualiser l'enchaînement des activités dans un domaine donné
- Cet artefact UP peut être constitué d'un ensemble de diagrammes d'activités UML

Chapitre IV : Modéliser le domaine

Chapitre IV : Modéliser le domaine

Diagramme de classes UML

- Un *diagramme de classe* décrit les types des objets qui composent un système et les différents types de relations statiques qui les relient
- Dans le contexte de la modélisation métier, ce diagramme sert de dictionnaire de données visuel

Chapitre IV : Modéliser le domaine

13 Diagramme de classes
- Introduction
- Classe
- Relations entre classes

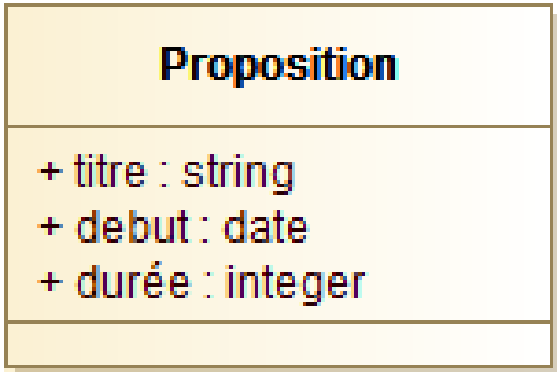
Classe

- Une *classe* est un classeur décrivant un type d'objets
- Une classe possède des *propriétés* et des *opérations* globalement nommées *caractéristiques*
- Dans le modèle du domaine, seules les propriétés sont utilisées

Notation

- Une classe est représentée comme un rectangle comportant plusieurs compartiments
- Le premier compartiment montre le nom de la classe, le second ses propriétés et le troisième les opérations
- Seul le compartiment du nom est obligatoire et d'autres compartiments peuvent être ajoutés (responsabilités, exceptions)
- Le nom de la classe débute par une majucule et peut être qualifié par des espaces de noms
- La classe peut être associée à des mots-clés entre accolades après son nom ($\{abstract\}$)

Exemple de classe



Une classe de l'application SIGS

Propriété

- Les propriétés décrivent les caractéristiques structurelles d'une classe
- Elles peuvent apparaître sur le diagramme sous la forme d'*attribut* ou de *terminaison d'association* (cf. section suivante)

Attribut

- Un attribut apparaît dans un compartiment d'une classe sous la forme d'une chaîne de caractères :

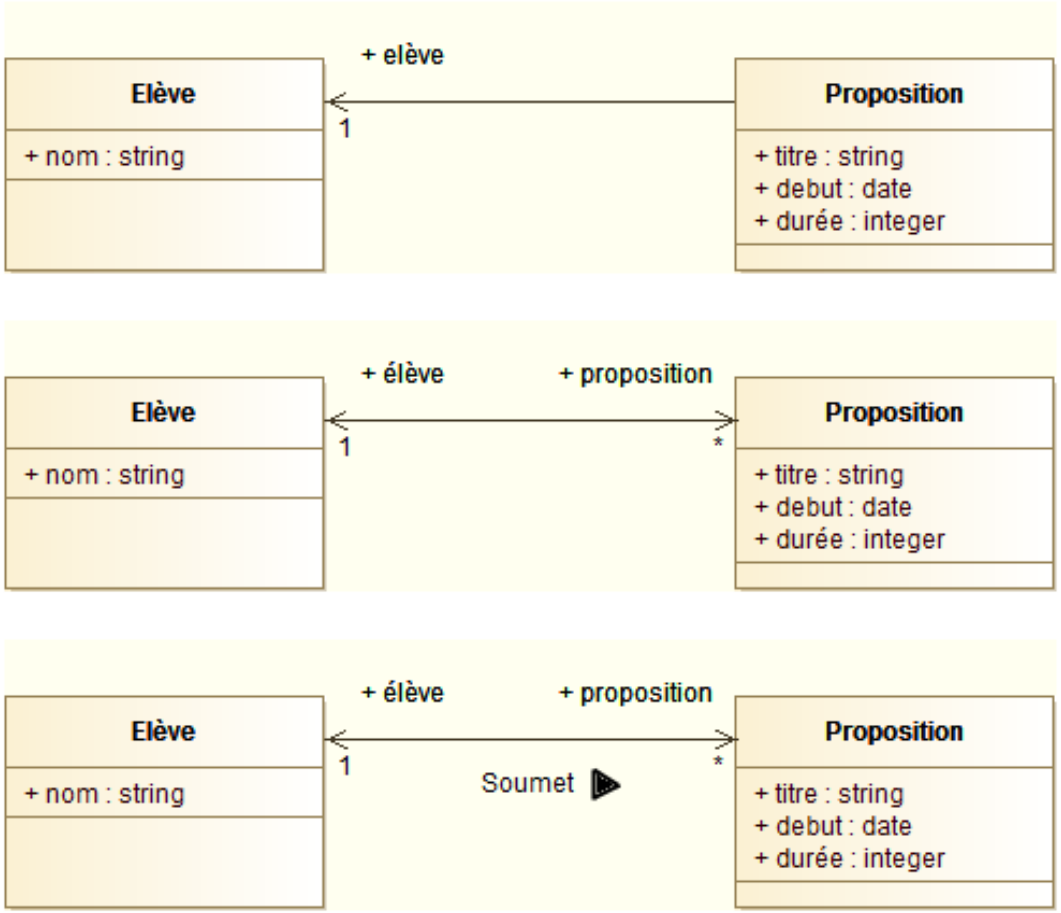
  `visibilité / nom : type multiplicité = valeurParDéfaut {mots-clé}`
- Seul le nom est obligatoire
- La *visibilité* contrôle l'accès à l'attribut : public (+), protected (#), package (~), private (-)
- Le caractère « / » précise que l'attribut est calculé
- La *multiplicité* précise le nombre d'instances de l'attribut
- Il est possible d'indiquer une borne inférieure et une borne supérieure pour la multiplicité (* pour illimité) ([0..1], [1], [1..*], ...)
- Les *mots-clé* modifient les caractéristiques de l'attribut ({*readOnly*}, ...)
- Un *attribut de classe* est représenté en soulignant l'attribut

Chapitre IV : Modéliser le domaine

Association

- Une *association* est un lien structurel entre deux (association binaire) ou plusieurs (association n-aire) classes
- Ce lien est visualisé comme un trait continu reliant les classes
- Une flêche pointe sur la classe cible (type de la propriété) et exprime la *navigabilité* de l'association
- La plupart des informations d'un attribut peuvent être utilisées sur une *terminaison* d'association (extrémité)
- Le nom et la multiplicité de la propriété apparaîssent près de la flèche du côté cible
- L'association peut être bidirectionnelle (flèche des deux côtés) et représente une paire de propriétés liées
- Une association peut aussi être nommée avec un groupe verbal et le sens de lecture peut être indiqué

Exemple d'association

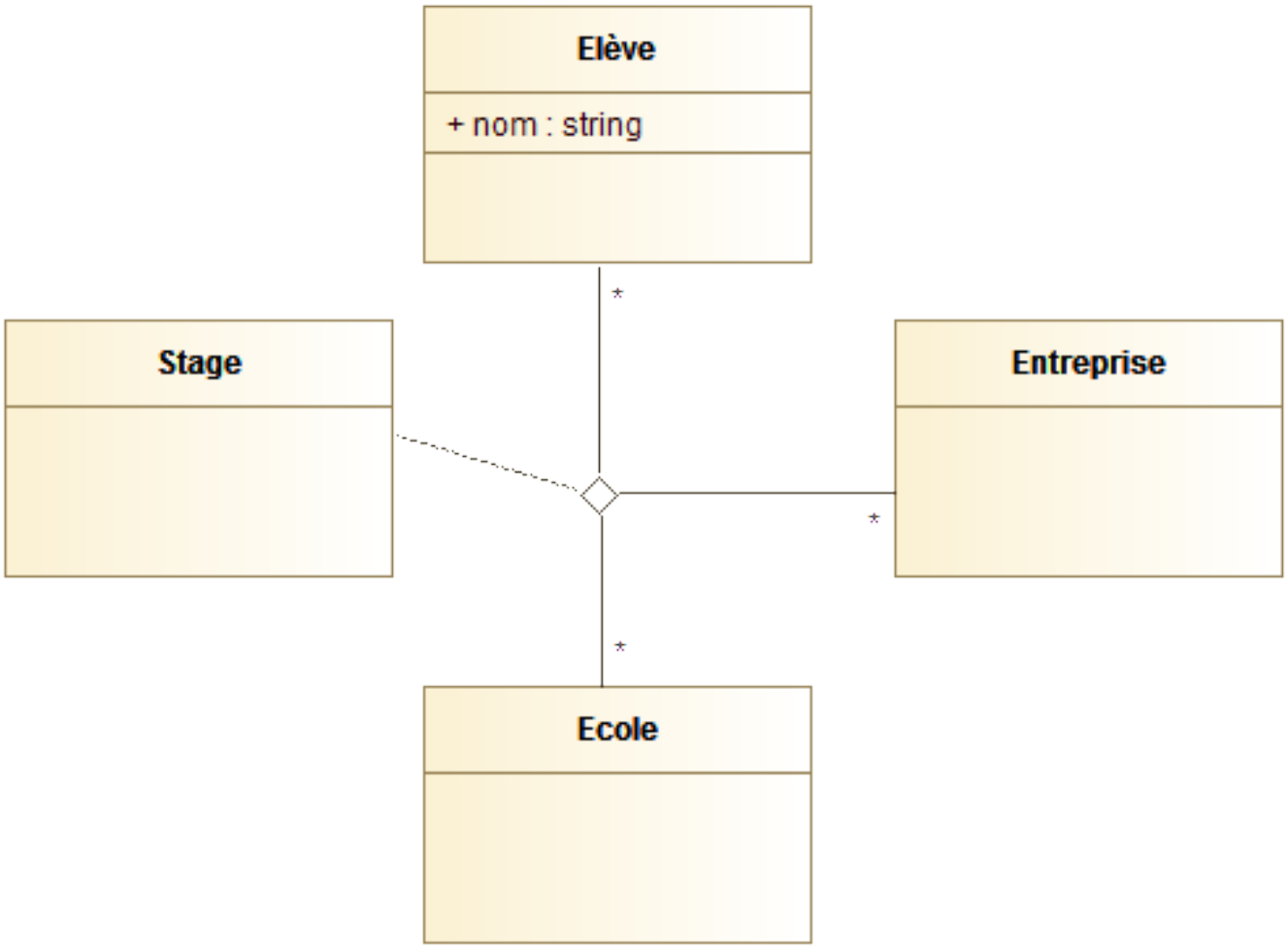


Représentation des associations

Association ou attribut

- Les attributs sont en général utilisés pour des propriétés simples (chaîne, date, booléen, . . .)
- Les associations relient plutôt des classes du domaine

Association n-aire et classe d'association



Représentation des associations n-aires

Interprétation de la multiplicité

- La multiplicité d'une terminaison d'association indique le nombre d'objets pouvant occuper la position de la terminaison
- Pour une association binaire, la multiplicité contraint le nombre d'objets de la classe cible, associé à un objet de la classe source
- Pour une association n-aire, la multiplicité d'une terminaison contraint le nombre d'objets de la classe liée à cette terminaison, associé à un n-1-uplet d'instances des autres participants
- La spécification de multiplicité (en particulier minimale) pour une association n-aire peut imposer des contraintes complexes à percevoir
  $\Rightarrow$ se limiter à une multiplicité minimale de 0 et maximale de 1 ou *

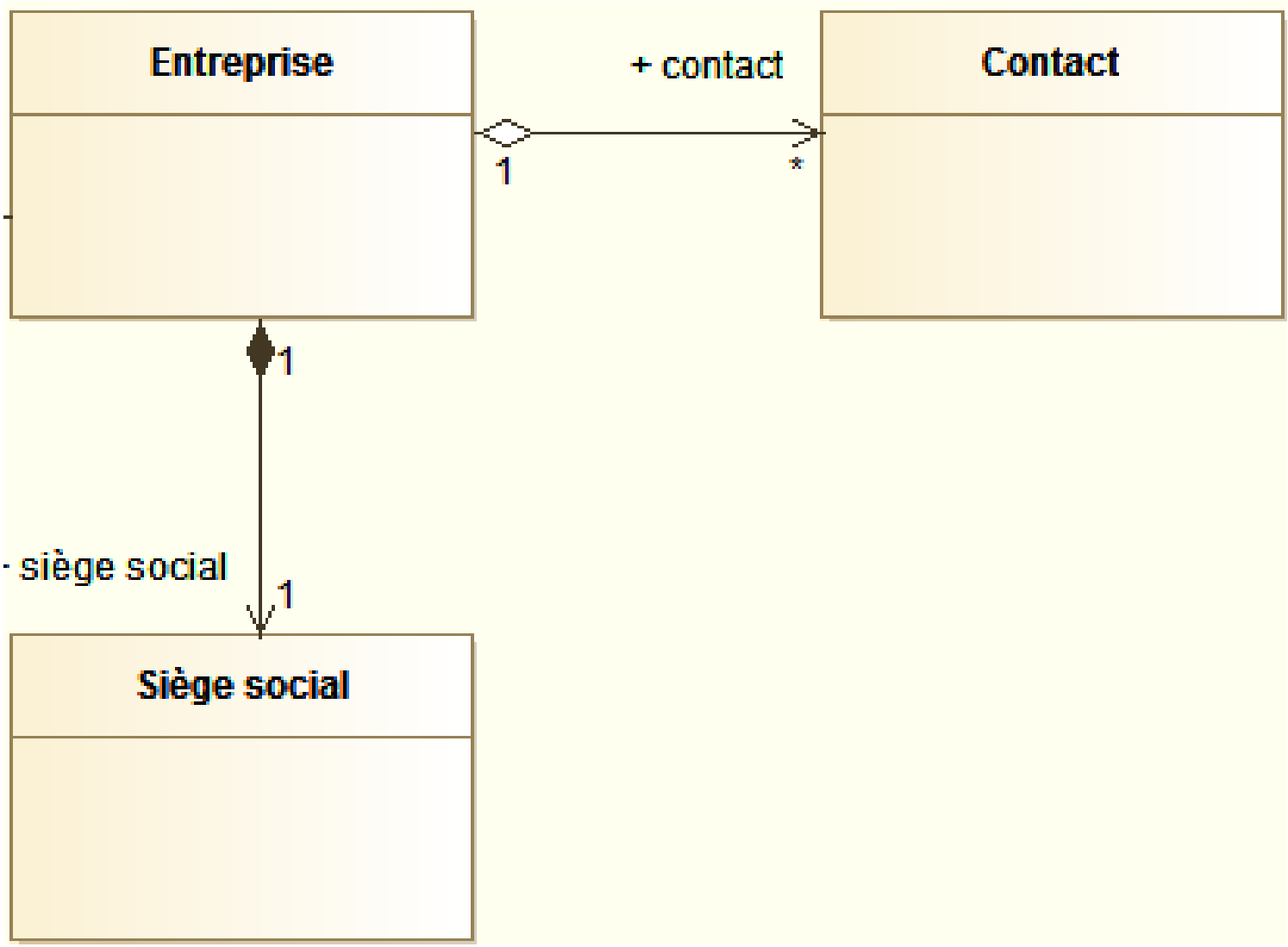
Multiplicité et cardinalité d'entité/association

- Pour une association binaire, la notation de la multiplicité est inversée par rapport à la notation de la cardinalité d'entité/association
- Pour une association n-aire, les sémantiques des deux notions sont différentes

Agrégation et composition

- L'*agrégation* et la *composition* représente une relation *tout/partie*
- Dans l'agrégation,
  - le cycle de vie de l'*agrégat* et des parties n'est pas lié,
  - la navigabilité et la multiplicité ne sont pas contraintes.
- Dans la composition,
  - le cycle de vie de l'*agrégat* et des parties est lié,
  - la multiplicité côté agrégat est d'au plus 1.
- Graphiquement, l'agrégation se représente par un losange vide et la composition par un losange plein du côté de l'agrégat

Exemple d'agrégation et de composition



Représentation de l'agrégation et de la composition

Chapitre IV : Modéliser le domaine

Chapitre IV : Modéliser le domaine

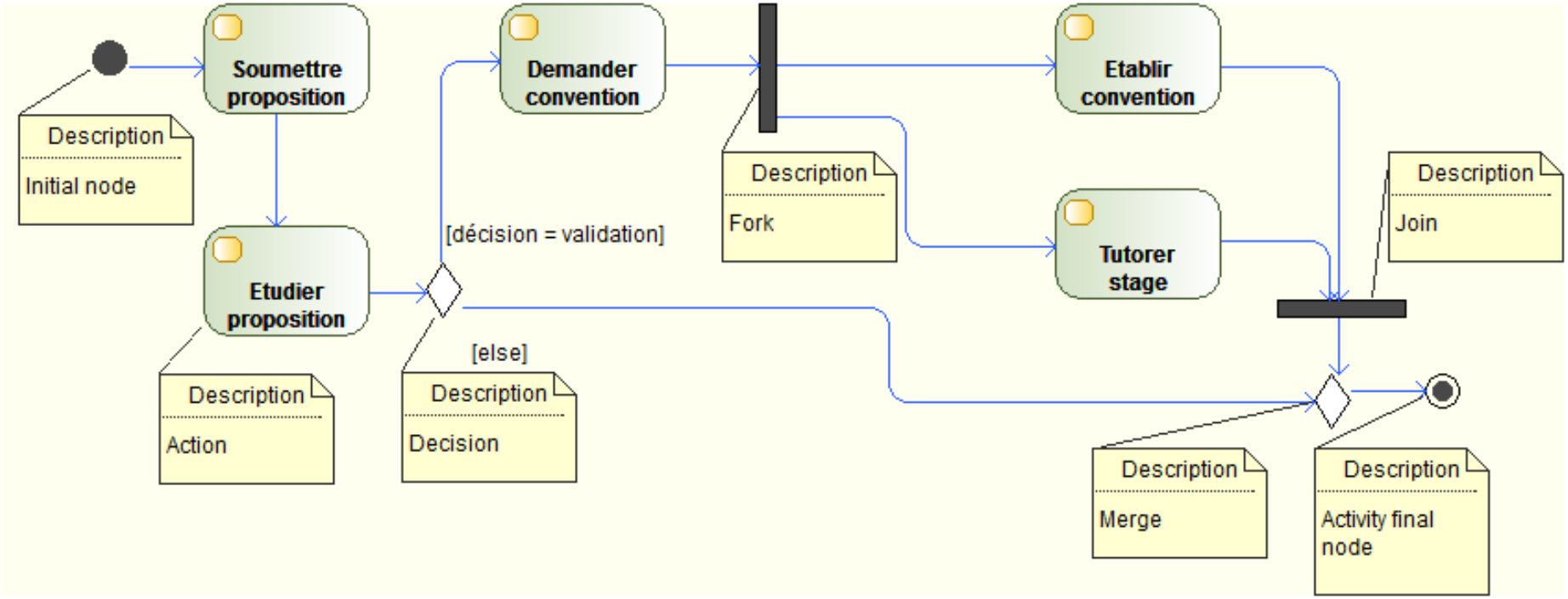
Diagramme d'activités UML

- Un *diagramme d'activité* UML visualise un modèle de traitements, i.e. l'enchaînement des activités d'un processus
- Il permet de représenter la séquence et le parallèlisme
- Il peut être utiliser pour modéliser des processus métier, des *workflows*, des flots de données ou des algorithmes complexes

Principaux concepts

Action	traitement élémentaire
Activité	séquence organisée d'actions
Transition	spécifie l'enchaînement des traitements (*flot de contrôle*)

Exemple de diagramme d'activités



Un diagramme d'activités pour l'application SIGS

Chapitre IV : Modéliser le domaine

14 Diagramme d'activités
- Introduction
- Flot de contrôle
- Flot de données
- Structuration des activités
- Interruption d'activités

Activité et action

- Une *action* est un traitement élémentaire (*call operation*, *create*, ...)
- Une action est représentée par un *nœud action*
- Une *activité* est un ensemble organisé d'actions et/ou de sous-activités
- Le flot d'exécution est modélisé par des nœuds reliés par des transitions
- Une activité est un *comportement*

Nœud et transition

- Un *nœud d'activité* représente graphiquement les phases d'une activités
- Il en existe trois types :
  - un *nœud action* visualise une action,
  - un *nœud objet* visualise les échanges de données,
  - un *nœud de contrôle* modélise le flot d'exécution.
- Une *transition* représente le passage d'une action à une autre
- Elle est déclenchée dès que l'action source se termine, provoque le début de l'action suivante et est instantanée

Nœuds de contrôle

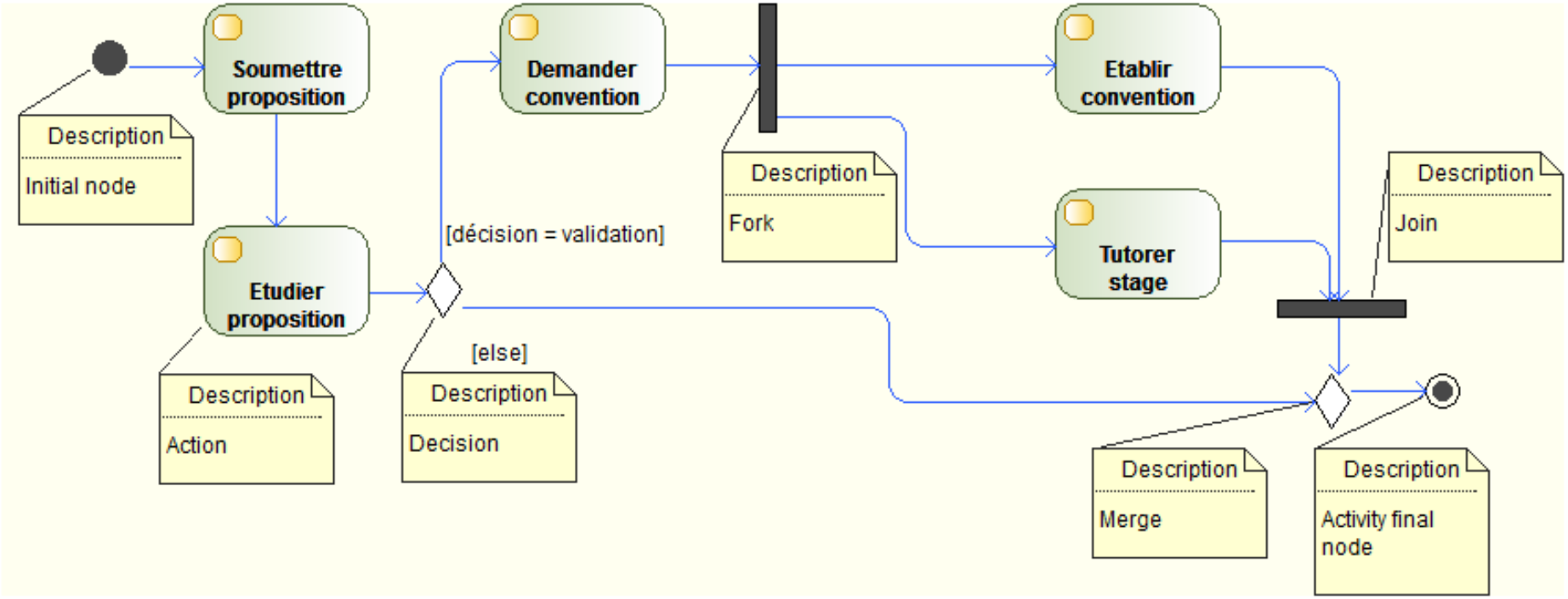
- Les nœuds de contrôle modifie le flot d'exécution d'une activité
- Il en existe plusieurs types :

initial	début d'une activité
fin d'activité	fin d'une activité
fin de flot	fin d'un flot d'exécution sans terminer l'activité
décision	choix entre plusieurs flots
fusion	regroupement de plusieurs flots
bifurcation	début d'activités en paralèlle
union	synchronisation de plusieurs flots

Représentaton graphique des nœuds de contrôle

initial	cercle plein
fin d'activité	cercle plein entouré d'un cercle vide
fin de flot	cercle barré d'un `X`
décision/fusion	losange vide
bifurcation/union	barre pleine

Exemple de diagramme d'activités



Un diagramme d'activités pour l'application SIGS

Nœud de décision et de fusion

- Un nœud de décision marque un choix entre plusieurs flots de sortie
- Chaque flot de sortie est conditionné par une *garde* (prédicat)
- Cette garde est évaluée pour savoir quel flot est franchissable
- L'ensemble des gardes doit normalement former une partition de l'ensemble des décisions possibles
  - si aucune transition n'est franchissable, le modèle est mal formé
  - si plusieurs sont franchissables, le choix est non déterministe
- Un nœud de fusion regroupe plusieurs flots alternatifs en un seul

Nœud de bifurcation et d'union

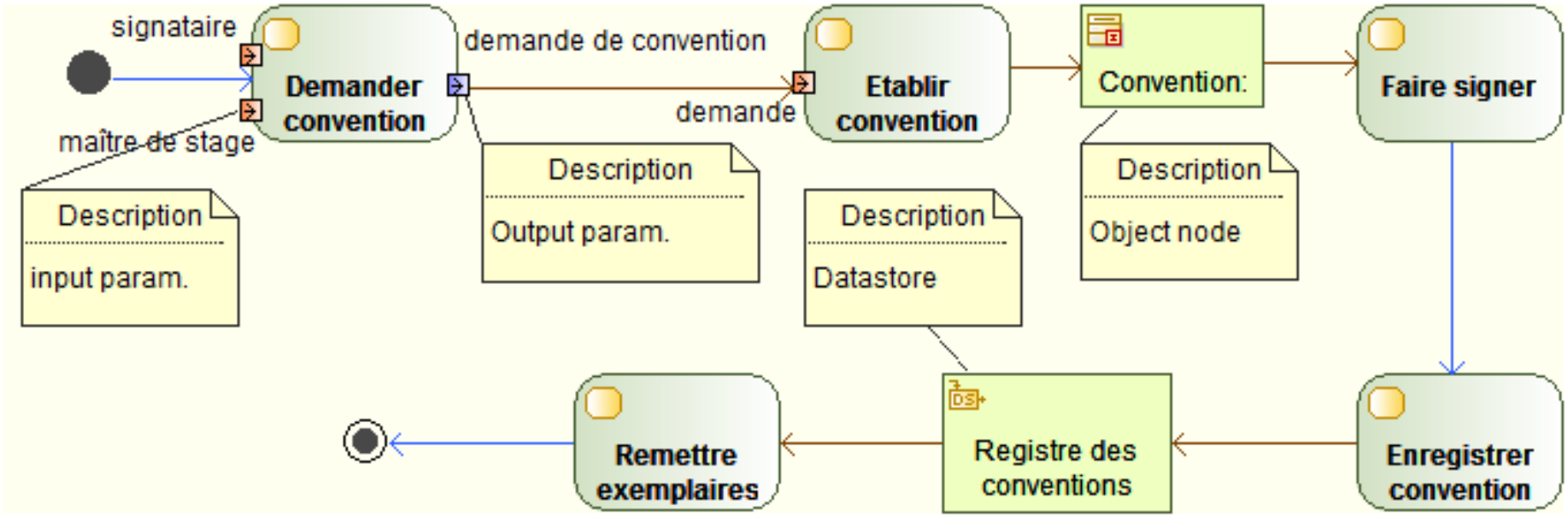
- un nœud de bifurcation (ou *débranchement)* sépare un flot en trant en plusieurs flots sortants concurrents
- un nœud d'union (ou *jointure*) synchronise plusieurs flots en trants concurrents

Chapitre IV : Modéliser le domaine

Flot d'objets

- Le flot d'objet représente explicitement les éléments échangés par les actions
- Sur le diagramme d'activité, on peut visualiser les paramètres d'une action ainsi que les objets échangés

Exemple de flots de données



Un diagramme d'activités pour l'application SIGS

Nœud objet

- Un nœud objet permet de représenter le flot d'objets échangés entre les actions
- Il possède un nom, un type, une éventuelle liste d'états et d'éventuelles contraintes
- Il peut être représenté comme
  - un paramètre d'entrée ou de sortie d'une action (petit carré sur la bordure d'une action),
  - un rectangle sur une transition précisant l'ensemble des informations associées
  - un tampon central (*central buffer*) qui accepte plusieurs entrées ou sorties d'objets
  - un stockage de données (*datastore*) qui est un tampon central assurant la persistance

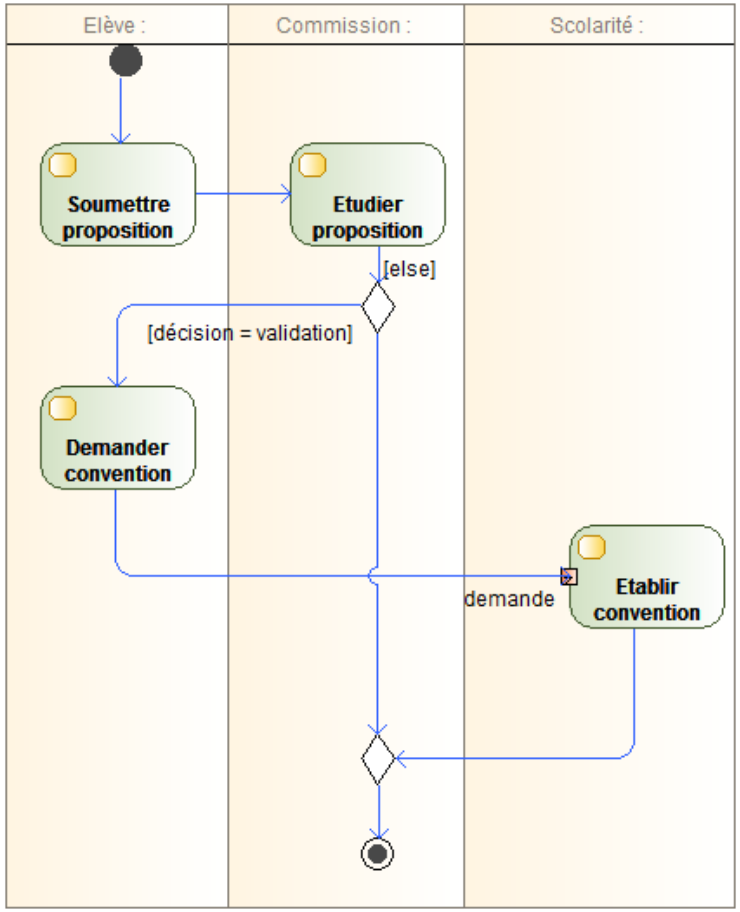
Chapitre IV : Modéliser le domaine

14 Diagramme d'activités
- Introduction
- Flot de contrôle
- Flot de données
- Structuration des activités
- Interruption d'activités

Partition

- Une *partition* (ou *couloir* ou *ligne d'eau*) permet de regrouper les nœuds d'activité
- Le regroupement peut se faire selon divers critères (responsable de l'action, ...)
- La partition peut être bidimensionnelle

Exemple de partitions

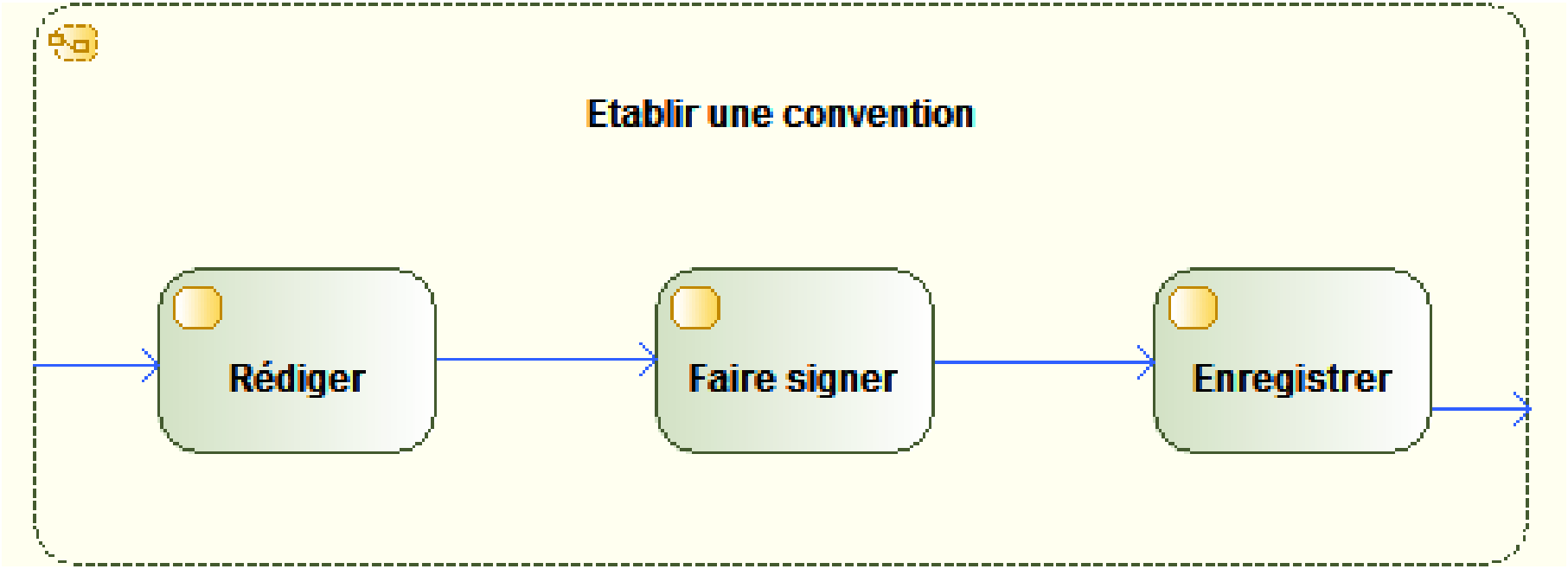


Un diagramme d'activités pour l'application SIGS

Nœud structuré

- Un nœud structuré représente une portion structurée d'une activité
- Graphiquement, c'est un rectangle en trait pointillé (stéréotype « `structured` »)
- Il en existe quatre types :
  - séquentiel (stéréotype « `sequence` »)
  - conditionnel (stéréotype « `conditional` »)
  - boucle (stéréotype « `loop` »)
  - zone d'expansion

Exemple de nœud structuré

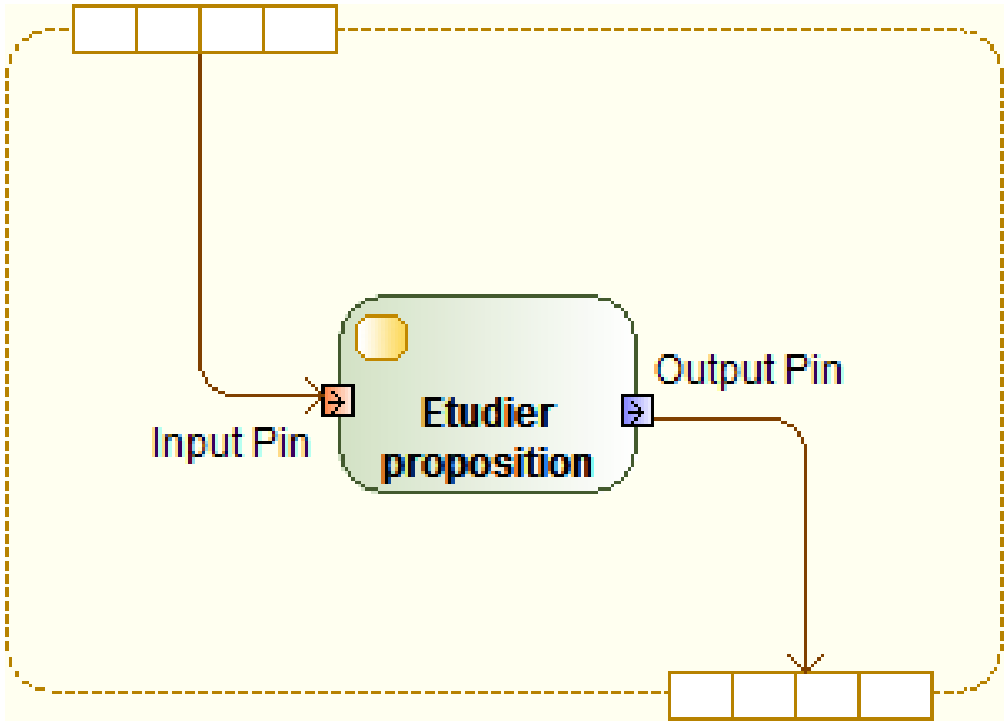


Un diagramme d'activités pour l'application SIGS

Zone d'expansion

- Une *zone d'expansion* représente une activité s'appliquant à une collection d'objets
- La collection est passée en paramètre de la zone d'expansion et une collection est produite en sortie
- Un stéréotype précise le fonctionnement de la zone (« `parallel` », « `iterative` », « `stream` »)

Exemple de zone d'expansion



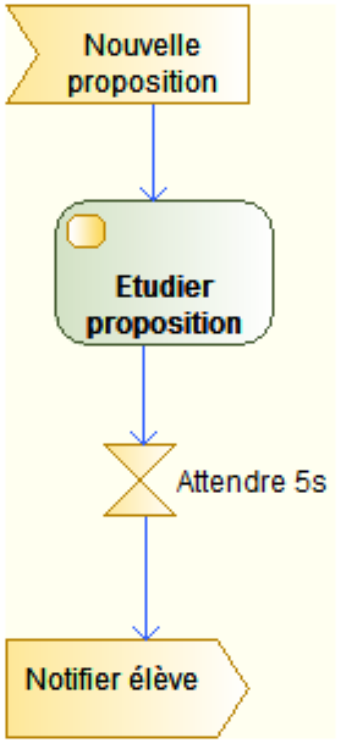
Un diagramme d'activités pour l'application SIGS

Chapitre IV : Modéliser le domaine

Signaux

- Des nœuds d'action spécifique permettent de manipuler des signaux
- Il en existe trois types :
  - réception d'un signal (*accept event*)
  - envoi d'un signal (*send signal*)
  - attente d'un évènement temporel (*accept time event*)

Exemple de signaux



Un diagramme d'activités pour l'application SIGS

Exceptions

- Les exceptions peuvent être représentées sur le diagramme d'activité
- Une activité pouvant lever un exception l'indique à l'aide d'un pin de sortie surmonté d'un petit triangle et précisant le type de l'exception
- Un gestionnaire d'exception est une action possédant un pin d'entrée du type de l'exception et les mêmes pins de sortie que l'activité surveillée
- L'activité surveillée est reliée au gestionnaire par un trait en zigzag ou surmonté d'un zigzag

Région interruptible

- Une *région interruptible* est une zone délimité d'une activité dont le flot peut être interrompue par un évènement nommé *interruption*
- Cette zone est délimitée par un rectangle pointillé aux coins arrondis
- Lorsque l'interruption se produit, l'exécution se poursuit dans un gestionnaire d'interruption
- l'émission de l'évènement et le gestionnaire sont reliés par un trait en zigzag

Chapitre V : Architecture de l'application

Chapitre V : Architecture de l'application

Définition

- L'architecture logique d'une application donne une vue globale de la conception d'un système
- Elle précise les modules dans lequels vont être créées les classes logicielles
- Elle n'implique pas de décision technique (OS, distribution,. . .)
  - les décisions techniques font parties de l'architecture de déploiement
- En général, une application OO est composée de plusieurs couches (interface, métier, . . .)

Principales couches d'une architecture logique

Présentation éléments de l'interface utilisateur (fenêtres, rapports, ...)

Application gestion des interactions avec la couche présentation, gestion des processus métier, transformation de données

Domaine règles et services du domaine

Infrastructure métier services (éventuellement externe) liés au métier

Services techniques services techniques de haut niveau (persistance, sécurité, journalisation, ...)

Fondation services et bibliothèques de bas niveau (I/E, threads, ...)

Architecture dans UP

- la spécification de l'architecture appartient au *Modèle de Conception* (discipline *Conception*)
- représentée sous la forme de diagramme de packages UML ou dans le *Document d'Architecture Logiciel*
- s'appuie sur les *Spécifications Supplémentaires*

Chapitre V : Architecture de l'application

Introduction

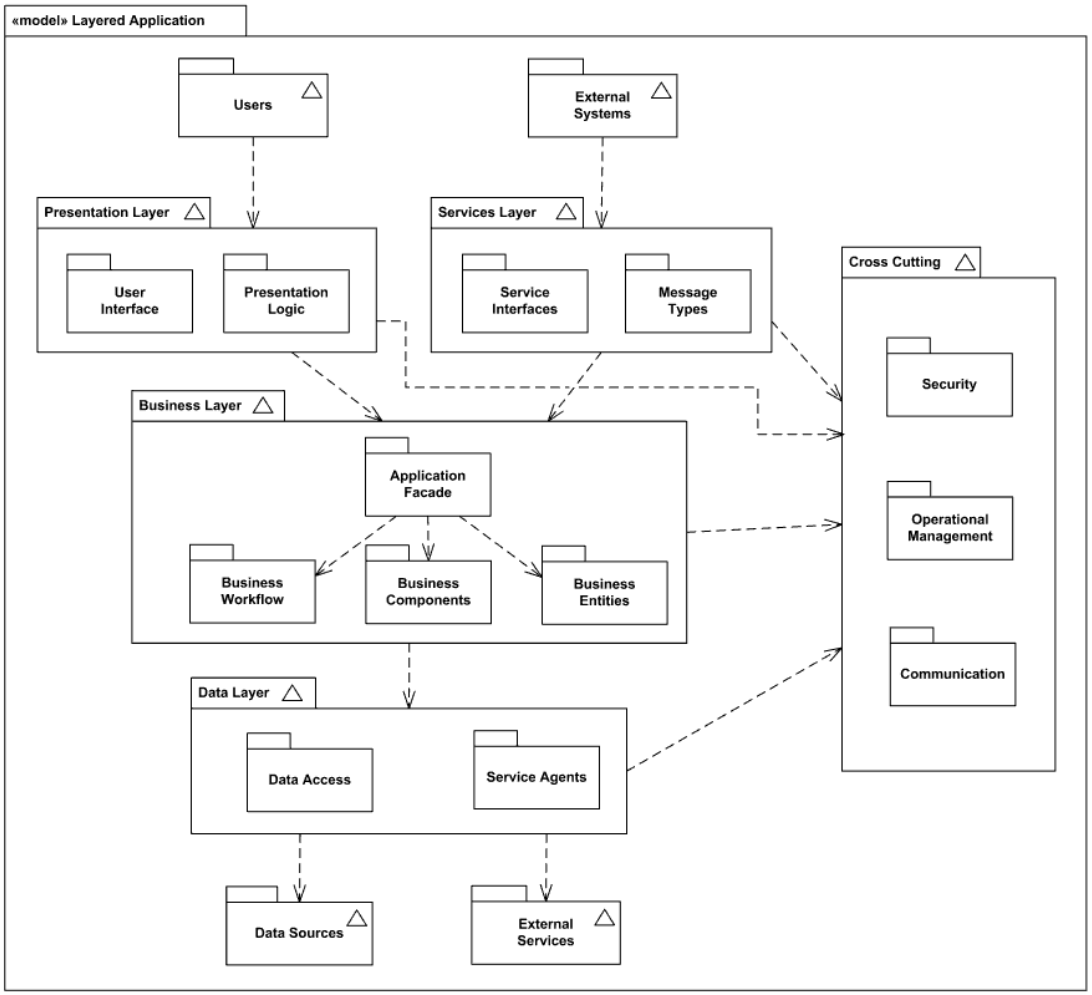
- Un diagramme de package permet de réprésenter les couches, sous-systèmes ou modules d'un logiciel
- Il montre le regroupement d'élément de modélisation (classes, cas d'utilisation, autres packages, ...)
- Il visualise également l'imbrication de modules

Notation

- Un package est représenté par une sorte de pochette à onglet
- Le nom du package apparaît soit dans l'onglet, soit directement sur la pochette
- Le diagramme de package montre également les *dépendances* entre package
- Une dépendance est visualisée comme une flèche pointillée (*ligne de dépendance*) partant du package dépendant vers le package dont il dépend
- Un package est aussi un *espace de nom*

Exemple

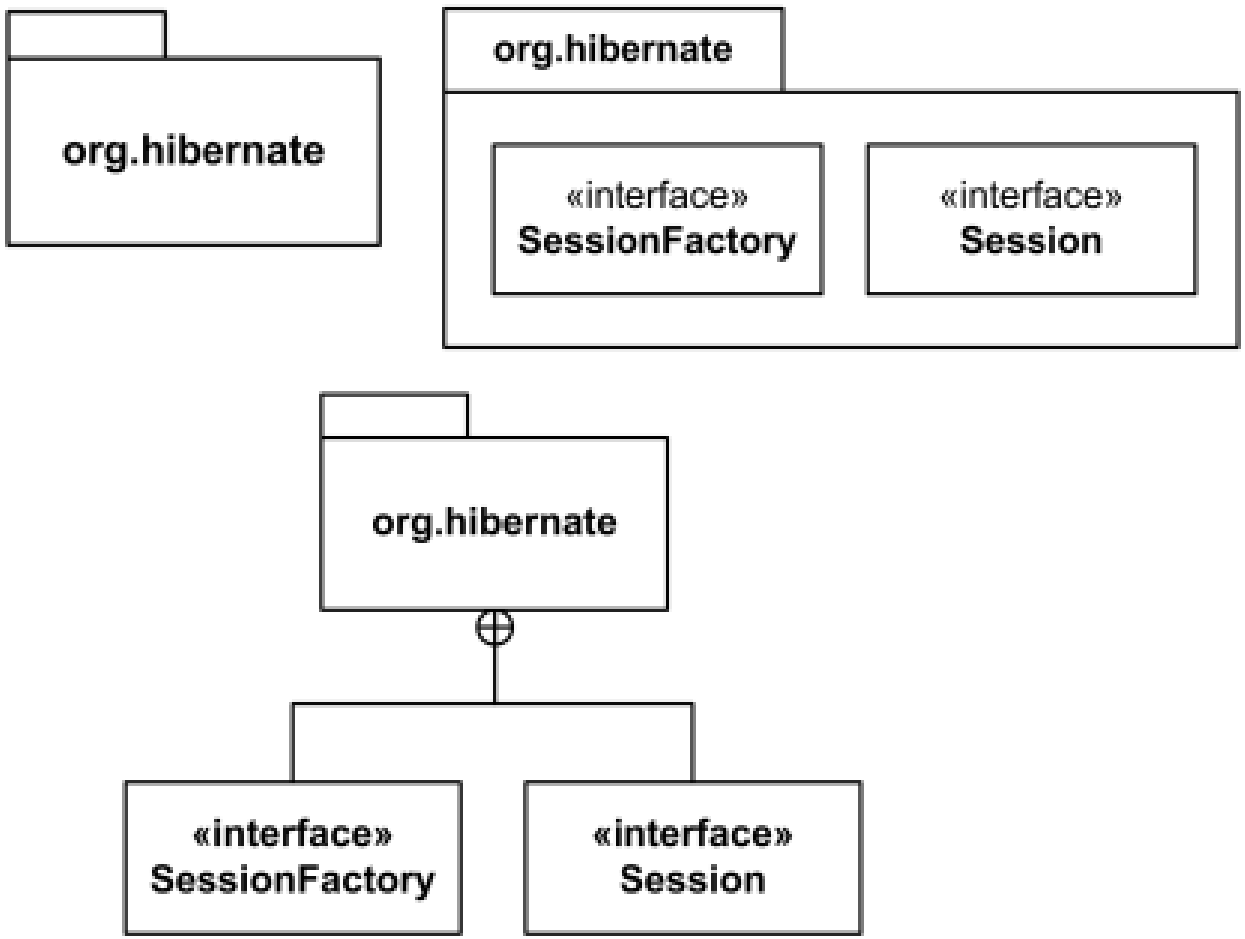
Un diagramme de packages pour une architecture en couche



source : Layered application model diagram

Exemple

Trois façons de représenter l'imbrication



source : Package org.hibernate

Chapitre VI : Conception OO

Chapitre VI : Conception OO

Définition

- La conception OO consiste à identifier les classes logicielles et à leur attribuer des responsabilités
- Différentes approches

  concevoir en codant utilise par exemple des outils comme le refactoring (approche TDD) et des diagrammes en mode esquisse

  tracer des diagrammes et coder UML est un support de discussion et/ou de documentation

  tracer uniquement les diagrammes approche de type MDE
- La question est de savoir quels diagrammes apportent une valeur ajoutée pour la conception OO

Modèle objet statique et dynamique

- Les modèles dynamiques comme les diagrammes d'interaction UML représentent la logique comportementale
- Les modèles statiques comme les diagrammes de classes ou d'objets UML visualisent la structure de l'application
- Le plus difficile et le plus intéressant est la réalisation du modèle dynamique (création d'objets, échange de message, . . .)
- Il faut pour cela identifier les responsabilités des classes logicielles
- Ces modèles sont en général réalisés en « parallèle »
- Tous ces modèles font partis du *Modèle de Conception* UP

Quelques techniques de conception OO

- Conception piloté par les responsabilités
  - réfléchir à la façon d'affecter des responsabilités aux objets
  - ce qu'un objet *sait* et ce qu'il *fait*
  - communauté d'objets responsables qui collaborent
- Principes GRASP (*General Responsibility Assignment Software Principles*)
- Modèles de conception GoF
- Cartes CRC (Classe-Responsabilité-Collaboration) fiches cartonnées (une par classe) détaillant les responsabilités et les collaborations

Chapitre VI : Conception OO

Introduction

- Un diagramme d'interaction représente un modèle dynamique
- Son rôle est de visualiser les objets interagissant par l'intermédiaire de messages
- UML en propose quatre types
  - diagramme de séquence
  - diagramme de communication
  - diagramme de vue d'ensemble des interactions
  - diagramme de temps

Remarque

Les diagrammes de séquence ont déjà été introduits dans la section sur les diagrammes de séquence système

Diagramme de séquence ou diagramme de communication ?

- Diagramme de séquence
  - \+ représentation claire de l'ordonnancement des messages
  - \+ notation riche
  - \- gestion du schéma difficile
- Diagramme de communication
  - \+ gestion du schéma simple
  - \- la numérotation rend plus difficile la lecture de l'ordre des messages
  - \- notation plus limitée

Chapitre VI : Conception OO

Ligne de vie

- Une ligne de vie (*lifeline*) représente un participant dans une interaction
- En général, une ligne de vie correspond à une instance d'une classe
- Une ligne de vie est représentée graphiquement par un rectangle dans lequel se trouve le libellé de l'élément
- Le libellé peut prendre différentes formes

  `: Convention` instance anonyme de la classe `Convention`

  `c1 : Convention` instance nommée `c1`

  `conventions : List<Convention>` instance de collection paramétrée

  `conventions[i] : Convention` instance sélectionnée dans la collection `collections`

- Dans un diagramme de séquence, le rectangle est connecté à une ligne vertical représentant le temps (de haut en bas)

Message

- Une interaction entre participants est représentée par un message
- Un message UML possède une syntaxe standard
  `retour = message (param : TypeParam) : TypeRetour`
- A part le nom du message, tous les éléments sont optionnels

Message

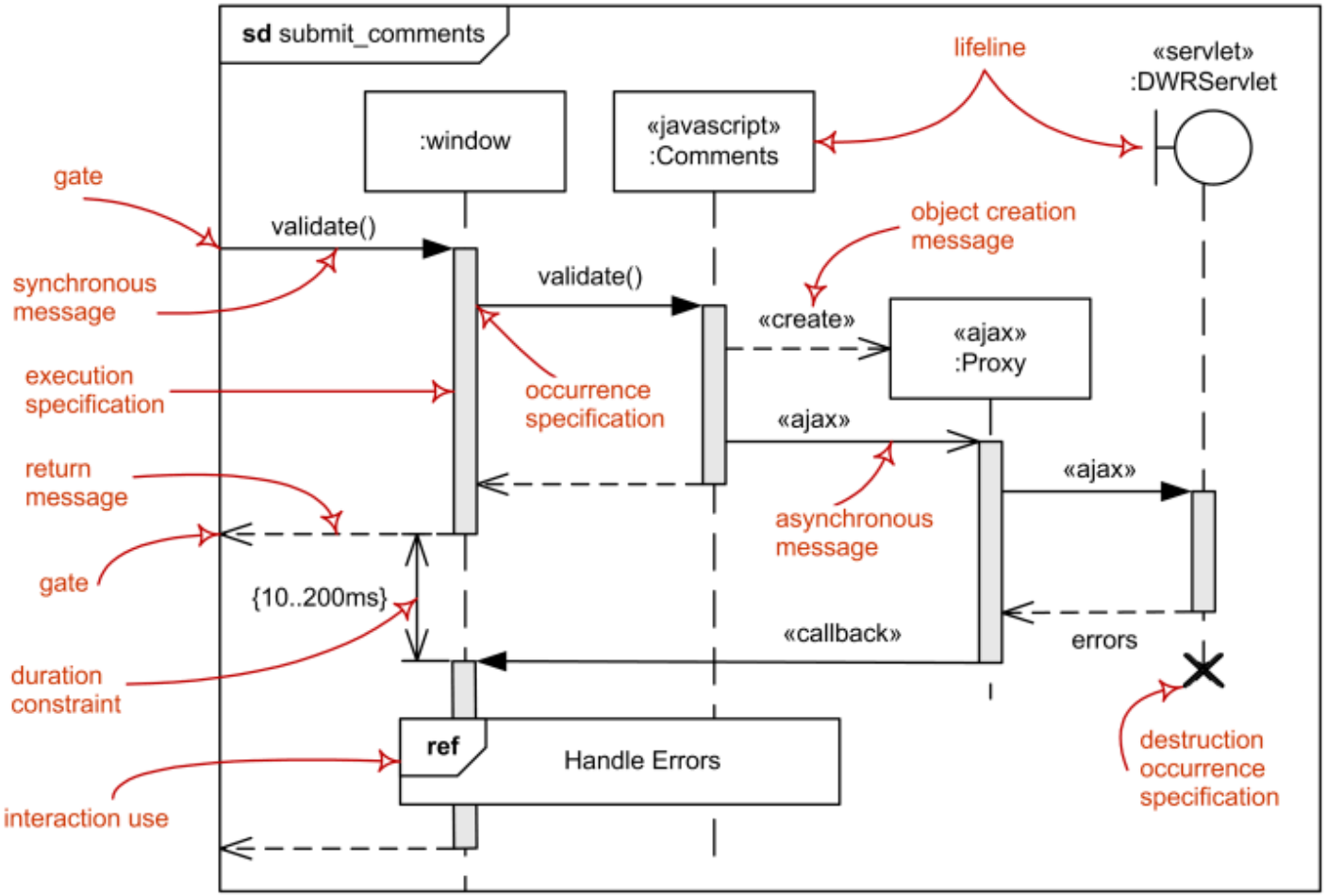
- Un message relie deux lignes de vie
- Un message est représenté par une flèche en trait plein dirigée vers la cible
  - pointe ouverte : message asynchrone
  - pointe pleine : message synchrone
- Un message synchrone peut être suivi d'une réponse (flèche en pointillée)
- L'invocation d'une méthode de classe a pour cible un participant instance de métaclasse (stéréotypé « `metaclass` »)

Spécification d'excution

- Une *barre de spécification d'exécution* marque l'exécution d'un traitement
- Elle est visualisée par un rectangle sur une ligne de vie

Exemple

Diagramme de séquence



Sequence diagram major elements.

Création et destruction de participants

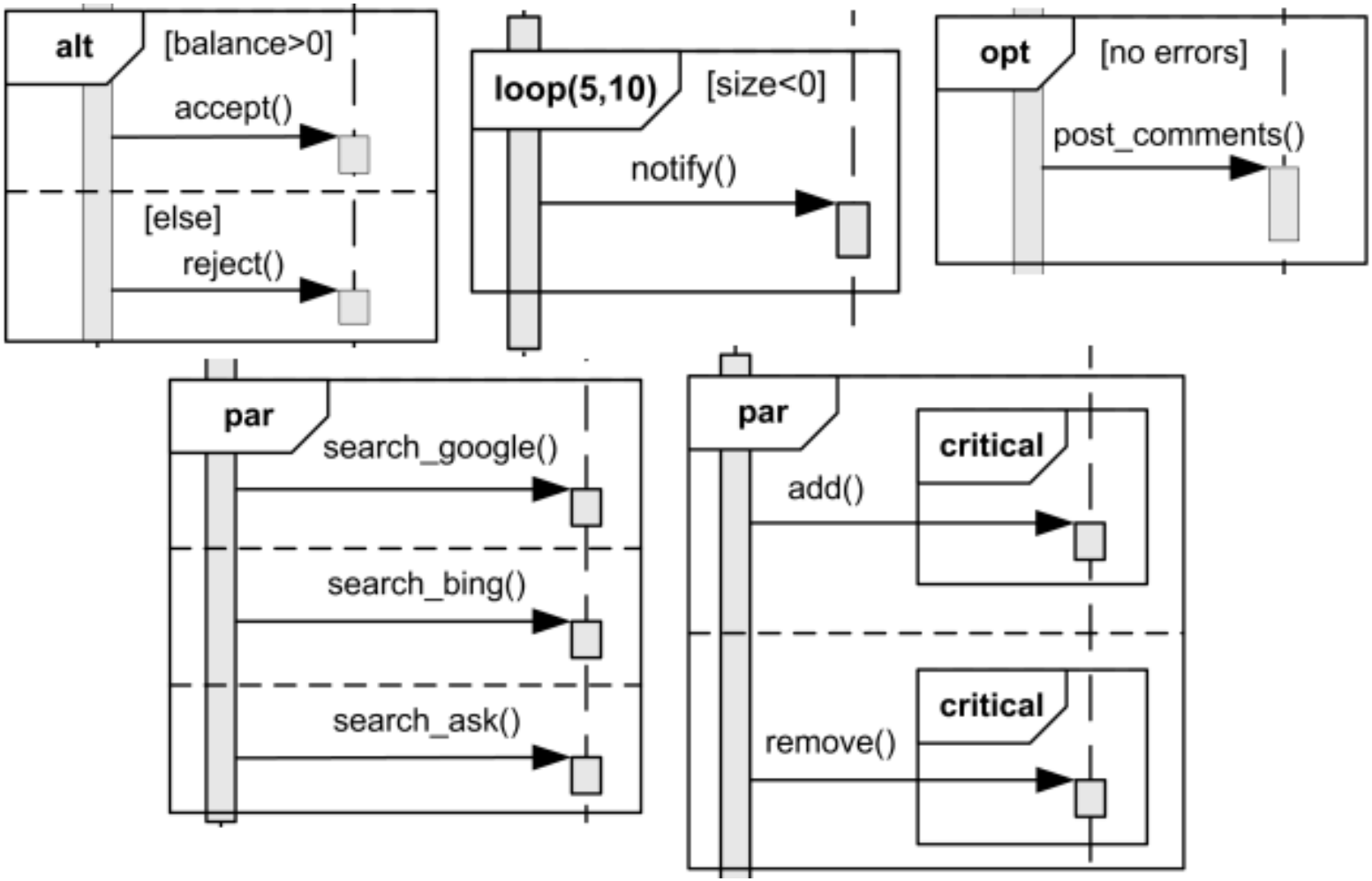
- Un message de création est représenté par une flèche pointillée
- La ligne de vie de l'objet créé est placé à la hauteur du message de création
- Un message de destruction est annoté par le stéréotype « `destroy` »
- Le moment de la destruction d'un participant et repéré par une croix sur sa ligne de vie

Cadre

- Un diagramme de séquence peut comporter des cadres
- Un cadre permet de regrouper ensemble des échanges de messages
- Un cadre comporte un *opérateur* et éventuellement une *garde*
- Les principaux opérateurs sont l'alternative (`alt`), la boucle (`loop`), le fragment optionnel (`opt`), le parallélisme (`par`) et la région critique (`region`)
- Les cadres peuvent être imbriqués

Exemple

Les principaux cadres



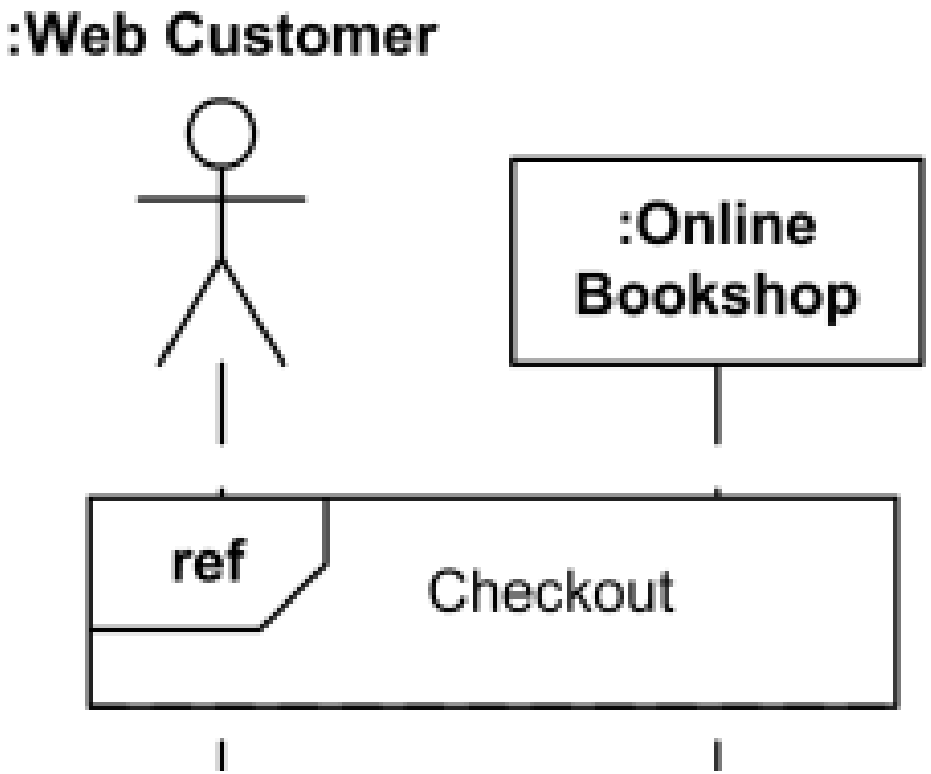
Combined fragments.

Utilisation d'interaction

- Une utilisation d'interaction est une référence à une interaction au sein d'une autre
- Les utilisations d'interaction s'appuie sur deux cadres
  - un cadre étiqueté `sd` suivi d'un nom et contenant un diagramme classique
  - un cadre étiqueté `ref` contenant le nom de l'interaction référencée

Exemple

Utilisation d'interaction



Interaction Use.

Lien

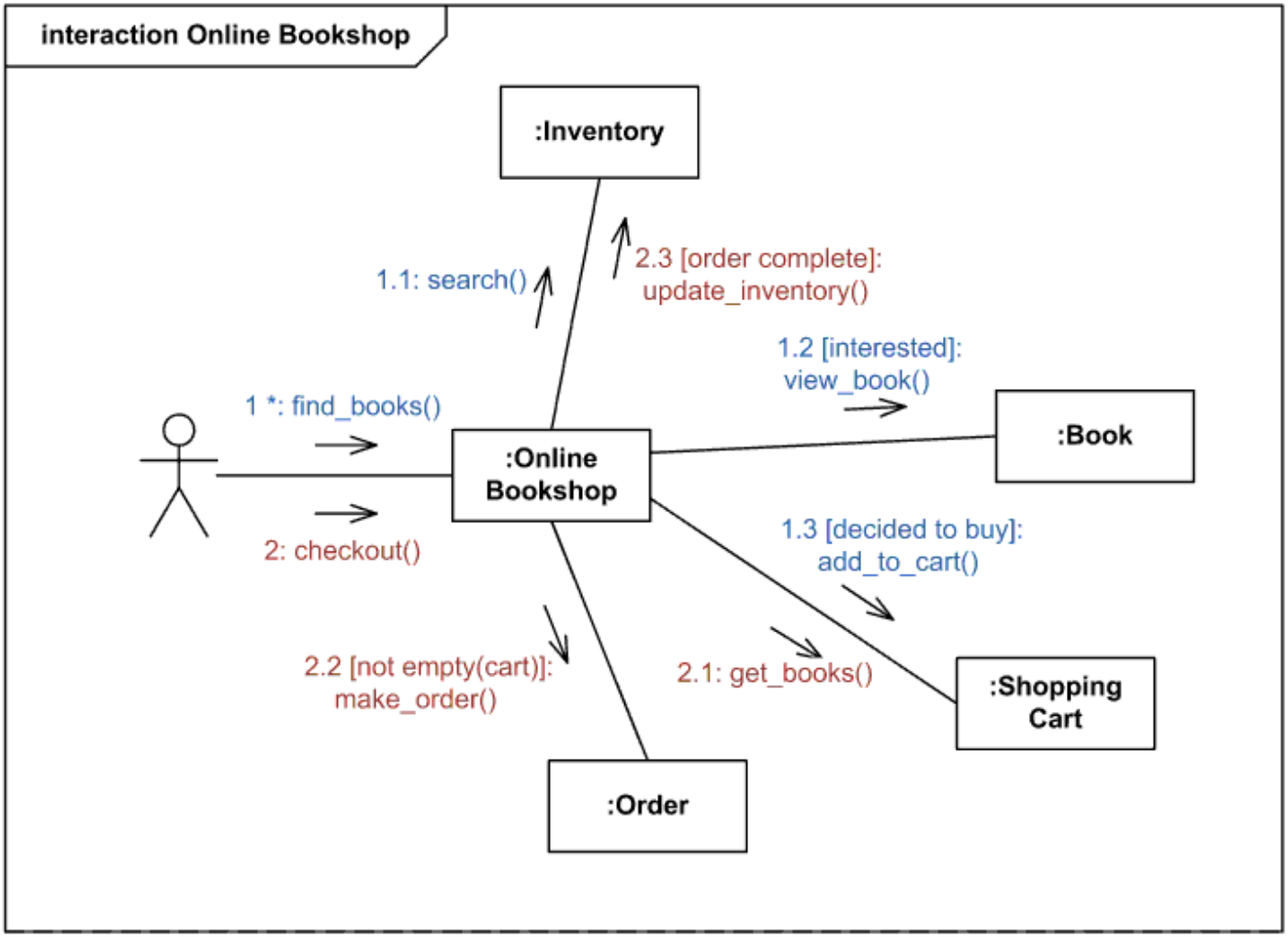
- Un lien est une connexion entre deux objets
- C'est une instance d'association
- Il est représenté par une ligne continue entre les lignes de vie
- Plusieurs messages peuvent circuler sur le même lien

Message

- Un message est une expression associée à une flèche pointant vers le destinataire
- Un numéro de séquence est associé à chaque message
- Chaque numéro précise l'ordre dans lequel les messages sont échangés ainsi que l'imbrication des appels
- Un message peut posséder une garde

Exemple

Diagramme de séquence



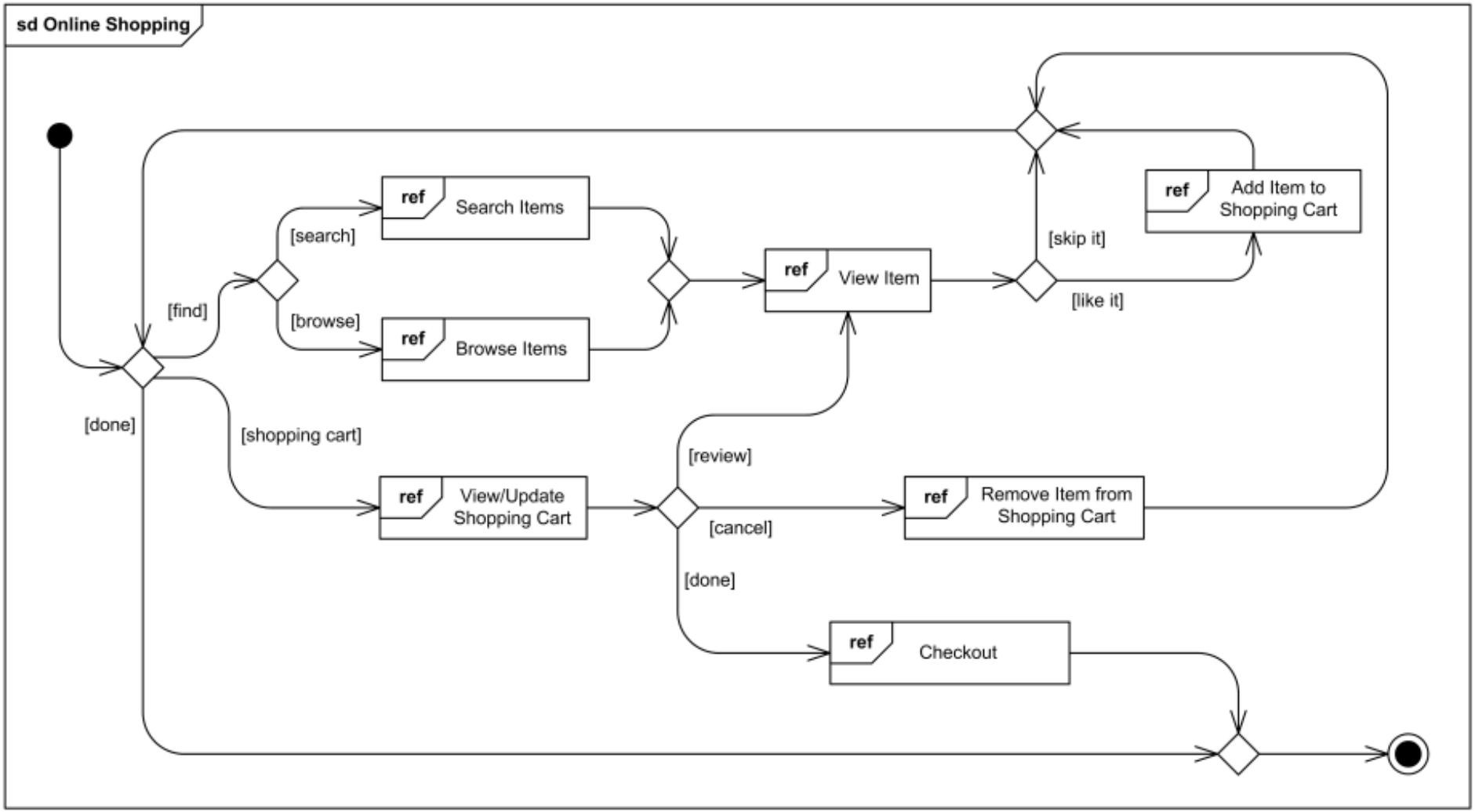
An example of communication diagram for online bookshop.

Diagramme de vue d'ensemble des interactions

- Ce diagramme représente le flot de contrôle entre les interactions
- C'est un diagramme d'interaction mais il est également proche du diagramme d'activité

Exemple

Un diagramme de vue d'ensemble des interactions



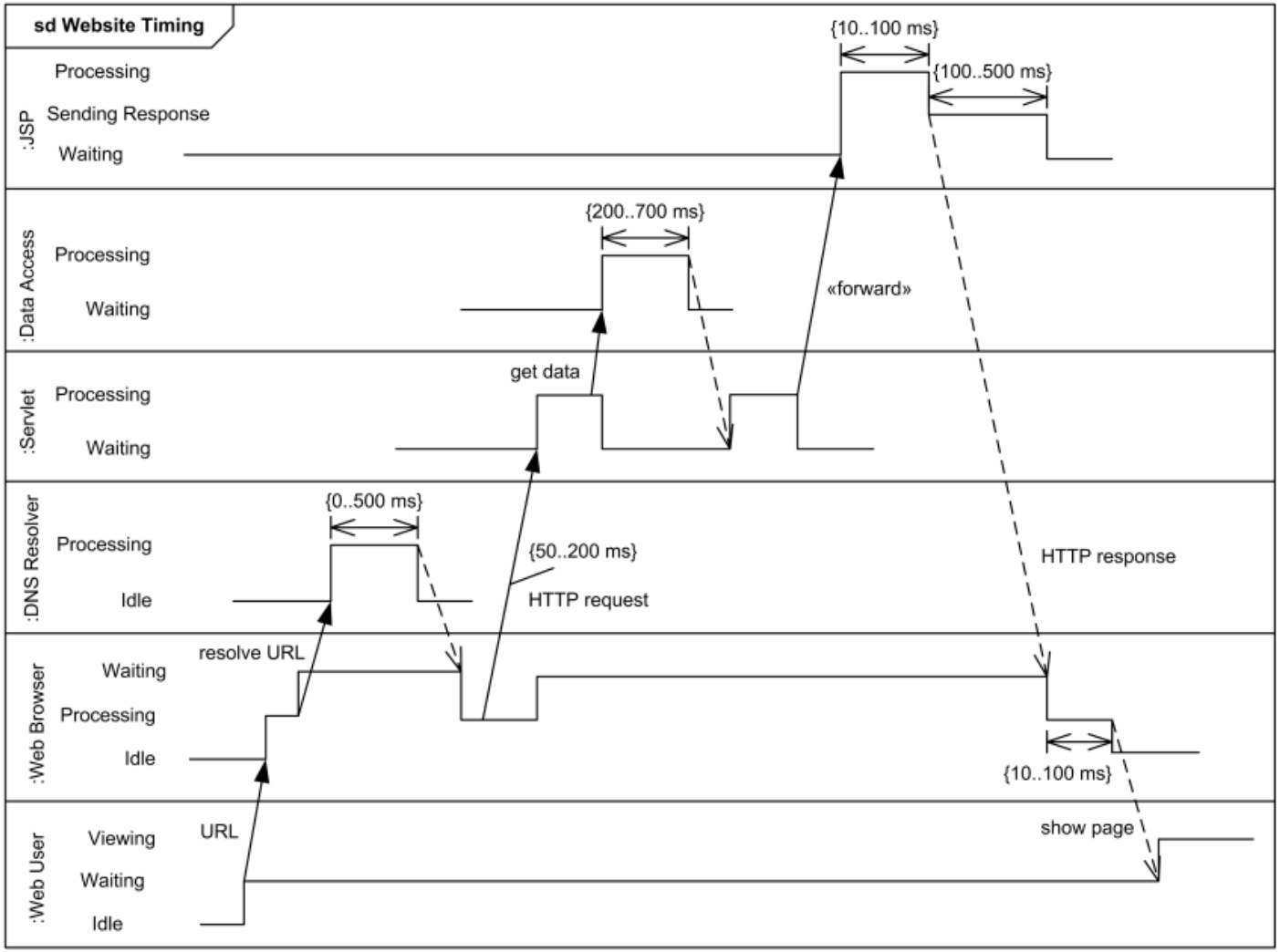
source : An example of interaction overview diagram for online shopping.

Diagramme de temps

- Le diagramme de temps met l'accent sur la représentation des interactions dans le temps

Exemple

Un diagramme de temps



source : User Experience Website Latency.

Chapitre VI : Conception OO

Diagramme de classes de conception

Remarque

Les diagrammes de classes ont déjà été introduits dans la section sur la modélisation du domaine.

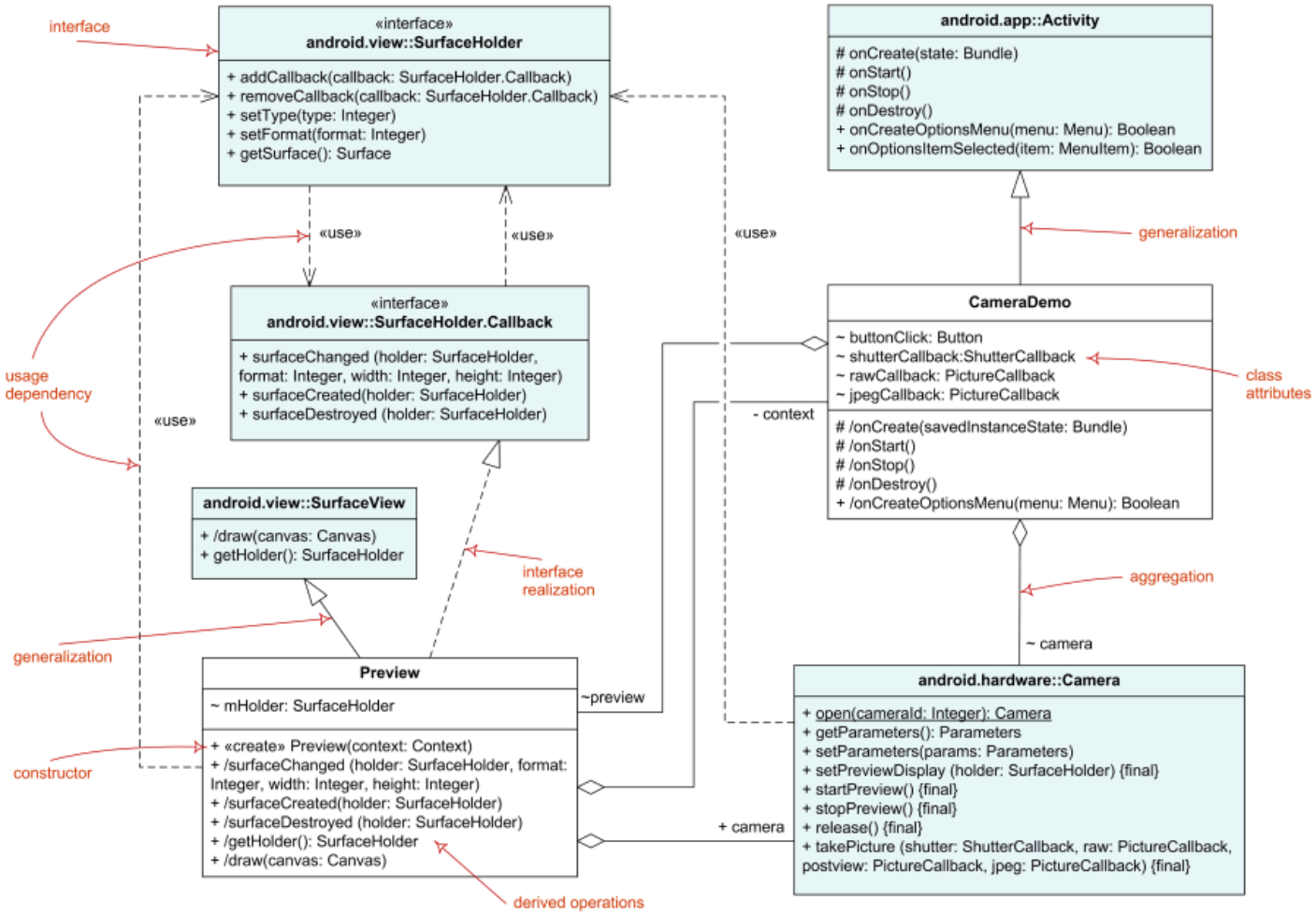
- Le *Diagramme de Classes de Conception* représente les classes logicielles d'une application

Opérations

- Le troisième compartiment d'une classe contient la signature des opérations supportées par la classe
- La syntaxe complète est la suivante :
  `visibilité opération (liste de param.) : TypeRetour propriétés`
- Les propriétés indiquent par exemple les exceptions lancées par l'opération, l'indication d'une opération abstraite, ...
- Une méthode est l'implémentation d'une opération

Exemple

Un diagramme de classes



source : Classes, interfaces, associations, usage, realization.

Généralisation, classes abstraites et polymorphisme

- La relation de généralisation est représentée par une ligne continue terminée par une triangle blanc pointant sur la classe la plus générale
- Ce lien exprime une relation taxonomique entre une classe plus spécifique et une classe plus générale
- Dans un diagramme de classes de conception, ce lien est assimilé à la notion d'héritage d'un langage de programmation
- Une classe ou une opération abstraite est annotée avec le mot-clé `{abstract}` ou écrite en italique

Interface

- Une interface est représentée comme un classeur avec le stéréotype « `interface` »
- La réalisation d'une interface par une classe est visualisée par une ligne pointillée terminée par une triangle blanc pointant vers l'interface
- Il est aussi possible d'utiliser une notation abrégée (*socket*)

Type de données

- Un *type de données* UML est un classeur dont les instances sont identifiées uniquement par leurs valeurs (*objet-valeur*)
- Un type de données est représenté par un rectangle avec le stéréotype « `dataType` »
- Un type de donées peut posséder des attributs et des opérations
- Un *type primitif* est une type de données utilisé pour représenter des valeurs atomiques (entier, flottant, . . .)
- Un type primitif est visualisé grâce au stéréotype « `primitive` »
- Une *énumération* est un type de données dont les valeurs sont définies dans le modèle
- Une énumération apparaît avec le stéréotype « `enumeration` » et peut posséder un compartiment listant les valeurs

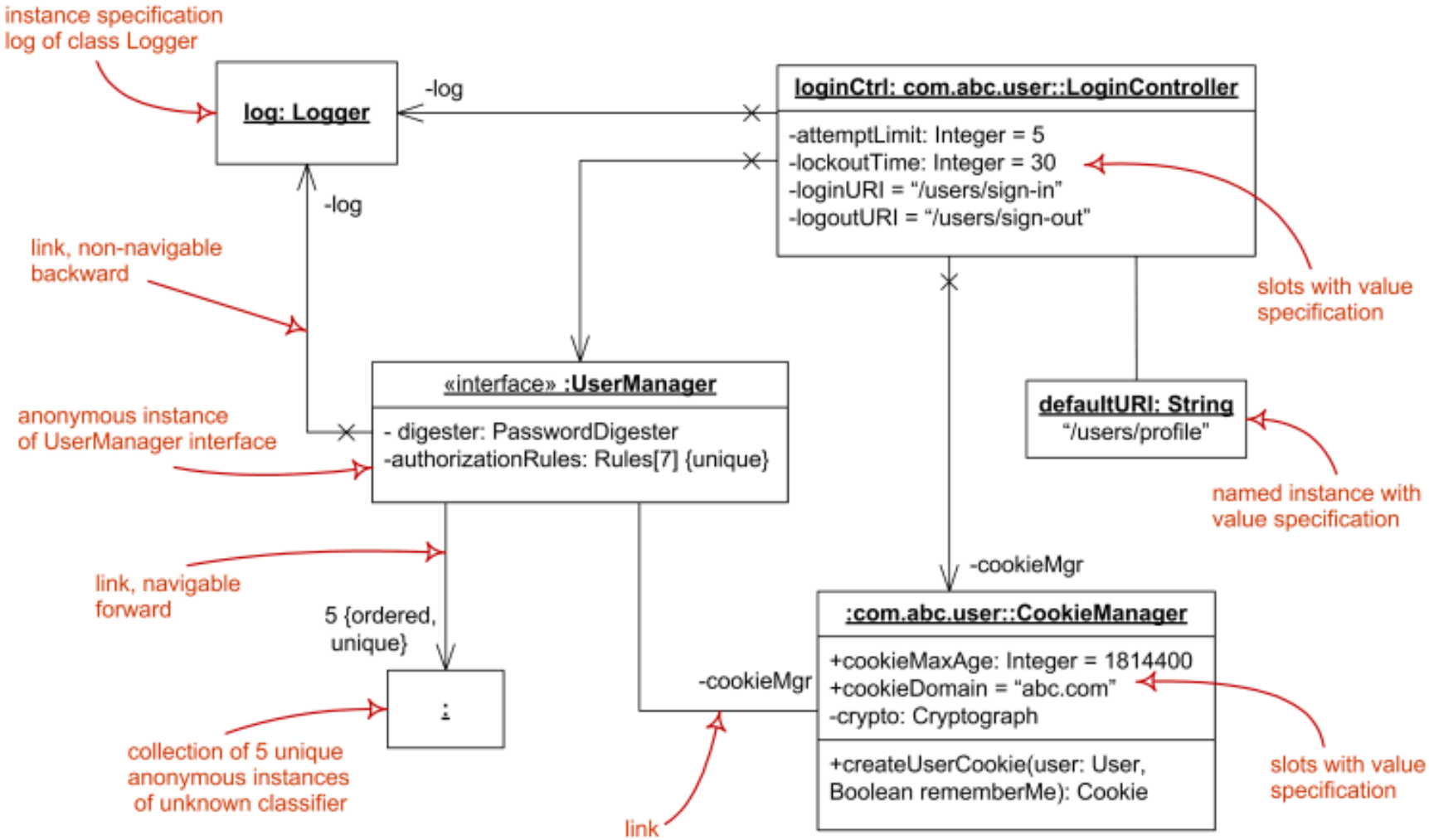
Chapitre VI : Conception OO

Diagramme d'objets

- Un *diagramme d'objets* montre des objets et les liens qui les relient
- C'est une instance d'un diagramme de classe
- Il permet de représenter le contexte d'une interaction, i.e. l'état de l'application avant ou après une interaction
- C'est un instantané de l'état d'un système à un instant donné
- Les objets sont représentés par des rectangles contenant le nom de l'objet souligné

Exemple

Un diagramme d'objets



source : Instance specifications, value specifications, slots, and links.

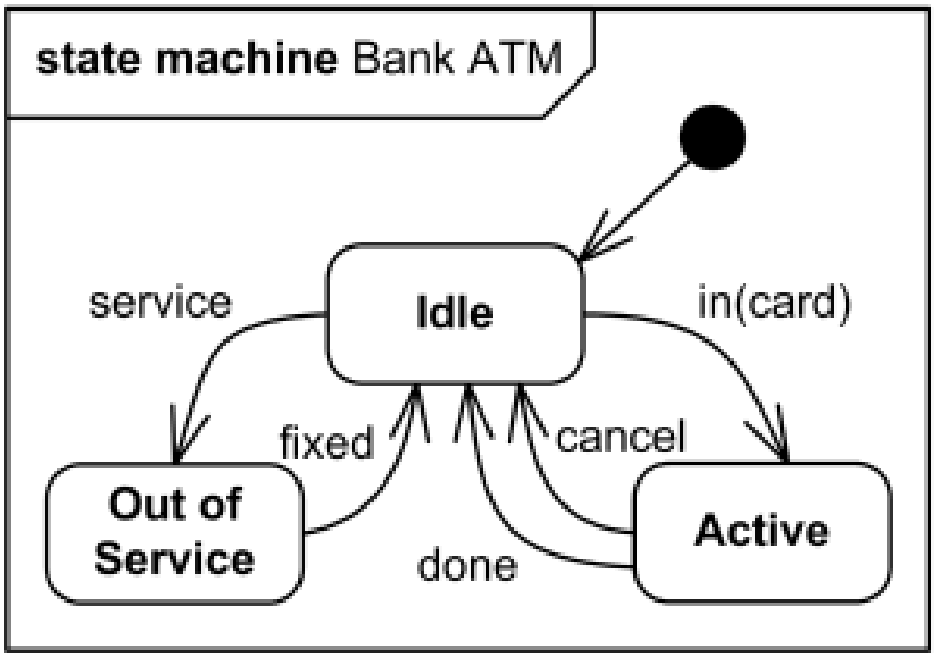
Chapitre VI : Conception OO

Diagramme de machine à états

- Une machine à états présente les états intéressants d'un objet ainsi que son comportement face à des événements
- Un état est visualisé par un rectangle aux bords arrondis
- Une machine à état comporte en général un pseudo-état initial et un état final (même notation que pour le diagramme d'activité)

Exemple

Un diagramme de machine à états



source : High level behavioral state machine for bank ATM.

Transition

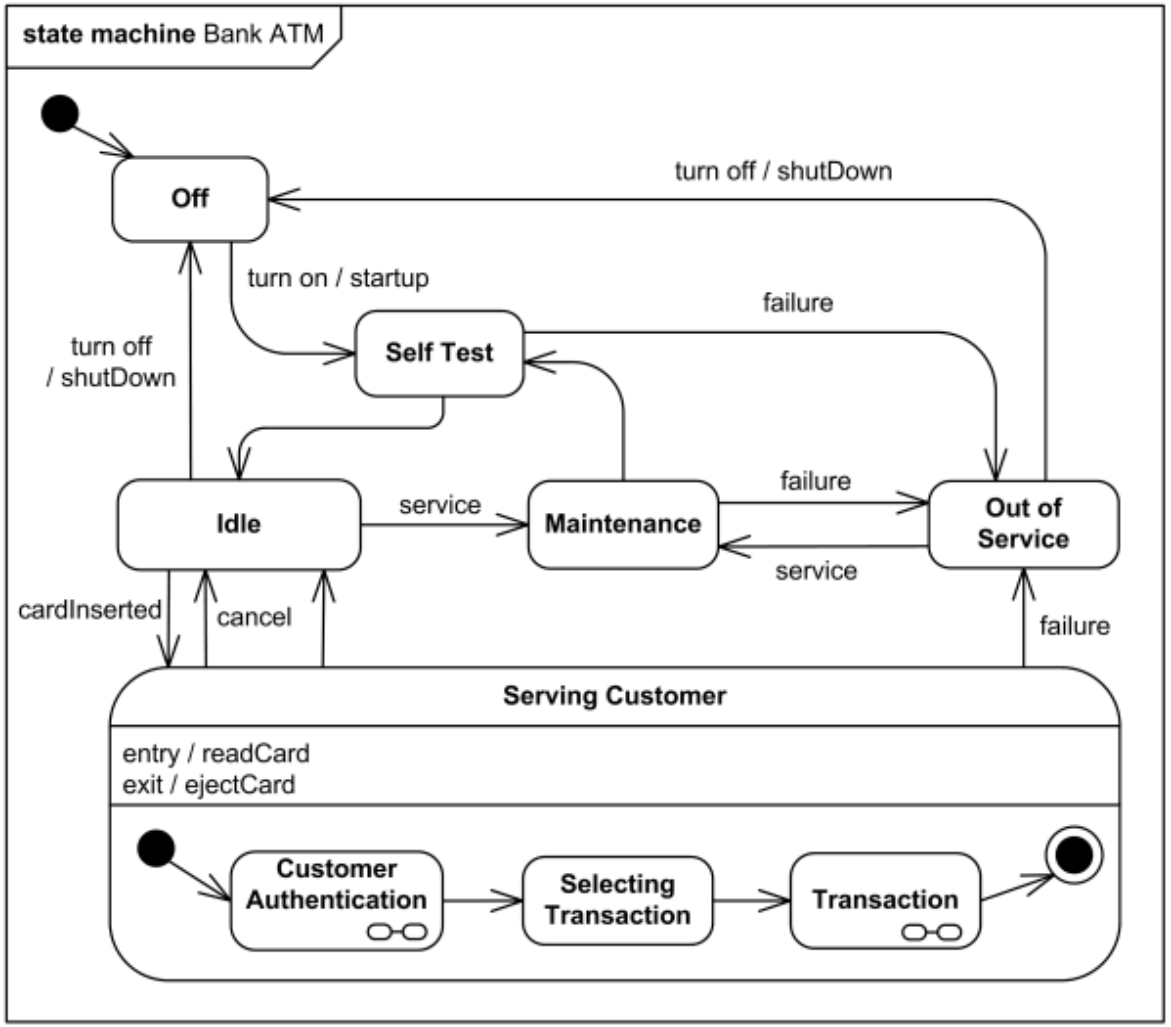
- Une transition représente un changement d'état de l'élément modélisé
- Elle peut comporter une étiquette (`libellé [garde]/activité`) dont chaque composant est optionnel
  - `libellé` est le nom de l'événement provoquant le changeent d'état
  - `garde` conditionne le changement
  - `activité` est un comportement qui se déroule lors du changement
- Une transition est représentée par une flèche entre deux états et porte le nom de l'événement qui l'a provoqué

Activité interne

- Un état peut réagir à des événements par l'intermédiaire d'*activités internes*
- Une activité interne est représentée par une chaîne de caractères dans un état (`libellé [garde]/activité`)
  - la signification est la même qu'une étiquette de transition
- les événements `entry` et `exit` permettent de déclencher une activité lors de l'entrée, respectivement la sortie, de l'état
- Un événement `do` précise que l'activité se déroule en continue durant l'état (*état d'activité*)

Exemple

Un diagramme de machine à états



source : Behavioral state machine example - Bank ATM.

Chapitre VII : Architecture de déploiement

Chapitre VII : Architecture de déploiement

Architecture physique

- L'architecture de déploiement d'une application représente son architecture physique
- Elle montre l'affectation d'artefacts logiciels à des machines physiques ansi que les communication entre ces artefacts
- Un diagramme de déploiement UML a pour rôle de représenter cette architecture physique
- Elle appartient à la Description de l'Architecture Logicielle de UP
- Le diagramme de composants quant à lui permet de représenter à grande échelle les éléments d'un système
- Le diagramme de composants est utilisé pour le *développement orienté composants* pour les *architectures orientées services* (SOA)

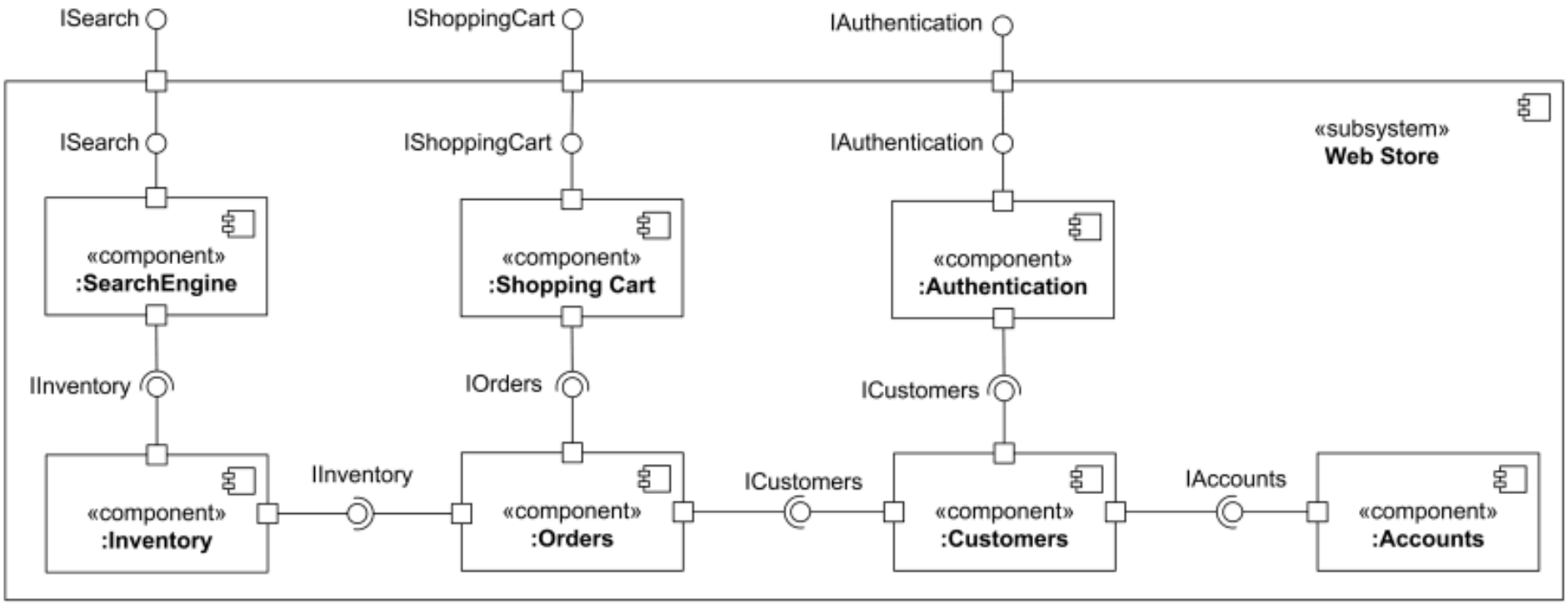
Chapitre VII : Architecture de déploiement

Composant

- Un composant est une partie modulaire d'un système qui peut être remplacée
- Il définit des interfaces fournies et requises
- En UML, il est possible de modéliser les composants avec des classes et des interfaces
- La modélisation par composants a pour objectifs
  - de mettre l'accent sur les interfaces,
  - d'insister sur l'aspect modulaire, autonome et remplaçable et sur la réutilisabilité.
- Un composant est représenté par un rectangle avec le stéréotype « `component` » ou un symbole spécifique
- Une relation de dépendance est utilisée pour montrer qu'un composant en utilise un autre

Exemple

Un diagramme de composants



source : An example of component diagram for a retail website.

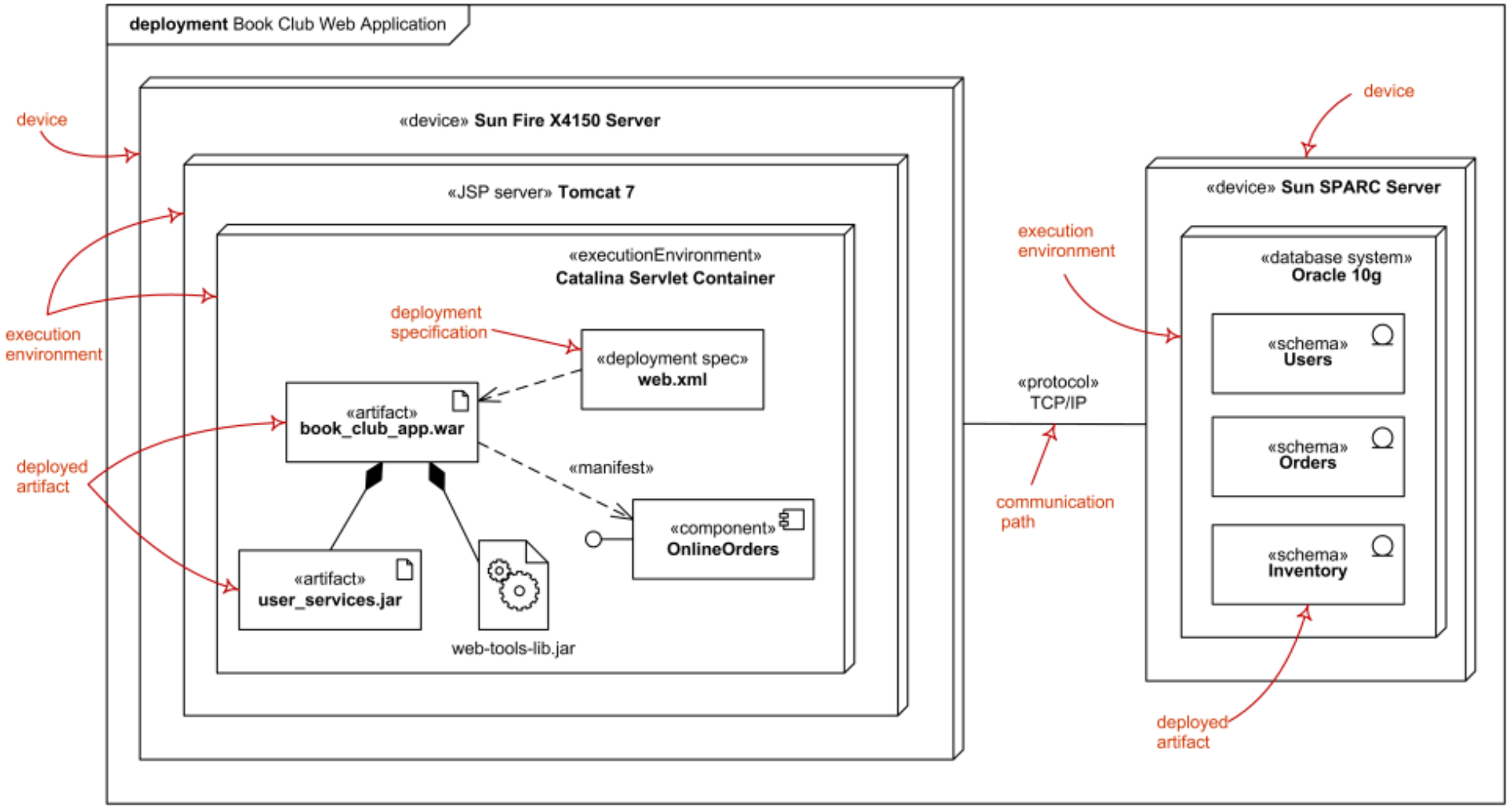
Chapitre VII : Architecture de déploiement

Introduction

- Un diagramme de déploiement montre la répartition des composants logiciels sur des composants physiques
- Un *nœud* est une unité susceptible d'héberger un logiciel
  - un *équipement* est un matériel
  - un *environnement d'exécution* est une plateforme d'exécution
- Un nœud héberge des *artefacts* représentants des produits du développement (exécutables, bibliothèques, fichiers de configuration, ...)
- Une *voie de communication* montre les nœuds communicant
- Le type de protocole peut être précisé comme label de la voie de communication

Exemple

Un diagramme de déploiement



source : An example of deployment diagram for web application.

Chapitre VIII : Le langage de contraintes OCL

Chapitre VIII : Le langage de contraintes OCL

exprimer des contraintes en UML

- Un contrainte est une restriction sémantique sur un élément de modèle qui limite les implémentations possibles d'un système
- UML permet d'exprimer un certains nombres de contraintes

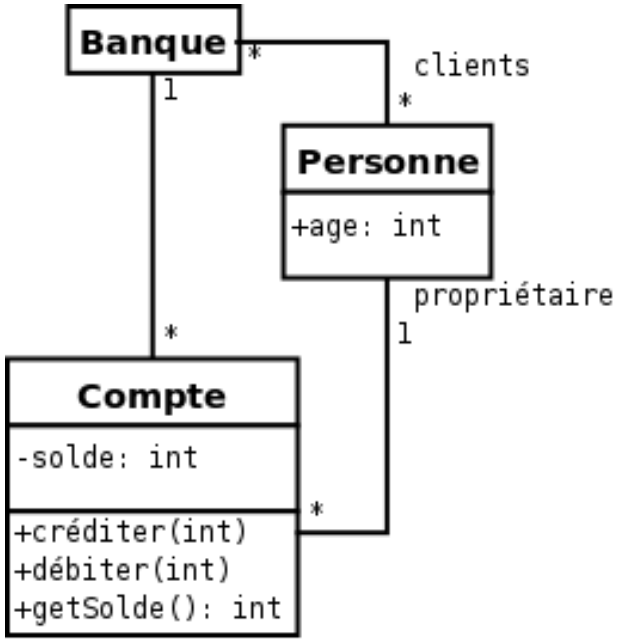
  structurelles comme les attributs, les types de relations ou la multiplicité

  de type comme les types des propriétés

  diverses comme la visibilité, les classes abstraites (« *abstract* »)
- En complément, il est aussi possible d'exprimer d'autres contraintes en langage naturel, avec un langage de programmation ou avec un langage formel
- Un contrainte apparaît dans un modèle entre accolades et est placée soit à proximité de l'élément concerné soit dans une note reliée
- OCL (*Object Constraint Language*) est un langage formel d'expression de contraintes adapté aux modèles UML

Exemple

Contraintes OCL pour une application bancaire



source : Exemple d'utilisation du langage de contrainte OCL sur l'exemple bancaire.

context Compte

inv : solde $>$ 0

context Compte :: débiter(somme : int)

pre : somme $>$ 0

post : solde $=$ solde@pre - somme

context Compte

inv : banque.clients -> includes (propriétaire)

Chapitre VIII : Le langage de contraintes OCL

Contexte d'une contrainte

- Le *contexte* d'une contrainte est l'élément de modèle concerné par cette contrainte
- Le contexte peut s'exprimer de deux façons
  - en plaçant la contrainte à proximité de l'élément entre accolades,
  - en utilisant la syntaxe `context <élément>` dans un document externe.
- On peut utiliser la notation :: pour qualifier le nom de l'élément

Invariant

- Un invariant exprime une condition toujours vérifiée pour un élément
- La syntaxe est `inv: <prédicat>`

Précondition et postcondition

- Une précondition exprime une condition qui doit être vérifiée avant un appel de méthode
- La syntaxe est `pre: <prédicat>`
- Une postcondition exprime une condition qui doit être vérifiée après un appel de méthode
- La syntaxe est `post: <prédicat>`
- Éléments particuliers utilisable dans l'expression
  - `result` est le résultat de l'opération
  - `<attribut>@pre` est la valeur de l'attribut avant l'appel

Résultat d'un opération

- La contrainte `body` permet de spécifier une expression décrivant le résultat d'une opération
- La syntaxe est `body: <expression>`
- Le type de l'expression doit correspondre au type de retour de l'opération

Définition d'attribut et d'opération

- La contrainte `let ...in` permet de définir des attributs pouvant être utilisés dans l'expression suivant le `in`
- La syntaxe est `let <attribut> = <expression> in <expression>`
- La contrainte `def` permet de définir des attributs ou des méthodes au niveau d'un classeur
- La syntaxe est `def <déclaration> = <expression>`

Initialisation et attribut calculé

- La contrainte `init` définit la valeur d'une propriété
- La syntaxe est `init: <expression>`
- La contrainte `derive` exprime la valeur d'un attribut calculé
- La syntaxe est `derive: <expression>`

Exemple

```
context Personne::marie : Boolean
init: false

context Personne::age : Integer
derive: Date::current().getYear() - dateNaiss.getYear()
```

Chapitre VIII : Le langage de contraintes OCL

Types et opérateurs prédéfinis

- OCL supporte un ensemble de types et d'opérations classiques sur ces types
- Les types prédéfinis sont `Boolean`, `Integer`, `Real` et `String`
- Parmi les opérateurs, on trouve
  - pour les booléens, `and`, `or`, `xor`, `not`, `implies` et `if ...then ...else ...endif`
  - pour les entiers et les flottants, `+`, `-`, `...`
  - pour les chaînes, `concat()`, `size()`, `...`

Type défini

- Tout type défini dans un modèle UML est utilisable avec OCL
- Pour les énumérations, la syntaxe est
  `<TypeEnuméré>::<constante>`

Collection

- OCL supporte les types de collection suivants

Set	ensemble au sens mathématique
Ordered Set	ensemble ordonné
Bag	collection non ordonnée
Sequence	collection ordonnée

Accès aux attributs, aux méthodes et navigation

- Un attribut ou une opération est accessible par son nom ou en le préfixant par `self`.
- La navigation sur une relation utilise la notation « . » préfixé par le nom du rôle

Exemple

```
solde
self.solde
self.getSolde()
self.propriétaire.nom
```

Opérations prédéfinies

- Les opérations suivantes sont applicables à tous les objets

oclIsTypeOf(<type>) exactement le même type ?

oclIsKindOf(<type>) le même type ou un dérivé ?

oclInState(<state>) dans l'état donné ?

oclIsNew() créé pendant l'opération ?

oclAsType(<type>) conversion de type

allInstances() retourne toutes les instances d'un type

Manipulation de collections

- L'opérateur `->` permet d'accéder aux caractéristiques d'une collection
- Quelques opérations supportées : `size()`, `includes(<object>)`, `isEmpty()`
- Quelques opérations sur les membres d'une collection
  - `<collection>->select(<condition>)` filtre la collection
  - `<collection>->forall(<condition>)` et `<collection>->exists(<condition>)`

Chapitre IX : Notions de métamodélisation

Chapitre IX : Notions de métamodélisation

Qu'est-ce qu'un métamodèle ?

- Un *métamodèle* définit les éléments utilisables dans un modèle
- Il précise la structure que doit avoir un modèle le respectant
- L'OMG utilise la notation des diagrammes de classes UML pour exprimer des métamodèles
- Un métamodèle ne décrit pas (ou peu) la sémantique d'un modèle
- Cette dernière est en général précisée en langage naturel

Intérêt de la métamodélisation

- Un métamodèle permet de décrire que qu'est un modèle valide
- Définir des traitements (transformation par exemple) au niveau du métamodèle permet de les appliquer à un ensemble de modèles

Modèles d'ordre supérieur

- On peut imaginer créer des métamétamodèles et des méta. . . métamodèles
- La question est de savoir où se trouve la limite
- Le principe consiste à faire en sorte qu'un niveau soit auto-descriptif, i.e. il se décrit lui-même
- La réponse de l'OMG est de définir quatre niveaux

Chapitre IX : Notions de métamodélisation

Chapitre IX : Notions de métamodélisation

Définition

- Un *métaformalisme* est un modèle d'un métamodèle, i.e. un métamétamodèle
- Il définit donc les concepts de base d'un métamodèle

Chapitre IX : Notions de métamodélisation

29 Métaformalisme
- Introduction
- MOF
- EMF

Meta Object Facility

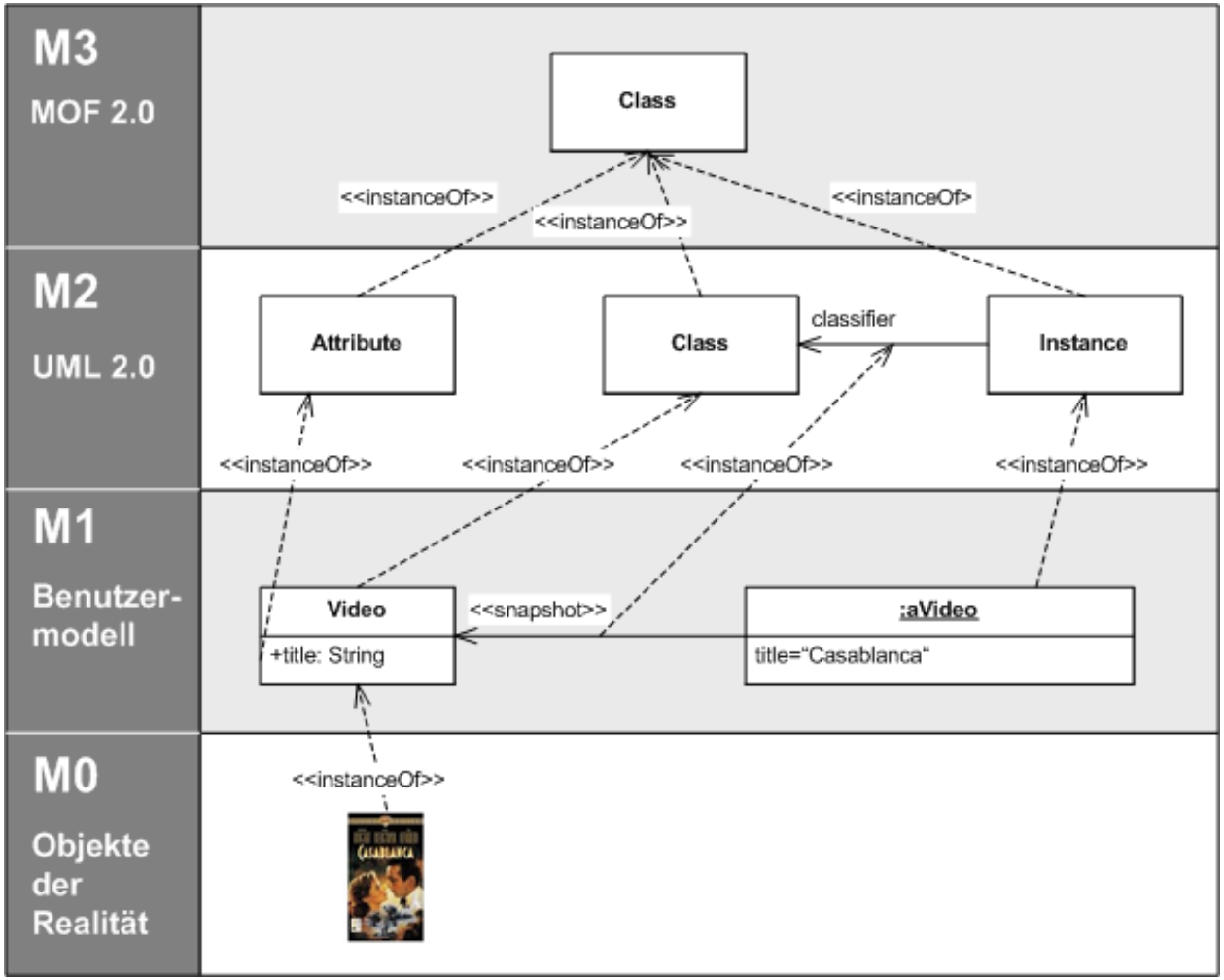
- Le *Meta Object Facility* (MOF) est le métaformalisme proposé par l'OMG
- Il est auto-descriptif
- Il définit les concepts de classe, relation, . . . utilisés dans les métamodèle
- C'est l'unique métaformalisme de l'OMG servant de base à tous les métamodèles

Niveaux du MOF

- L'OMG définit quatre niveaux de modélisation
  - M0 système réel
  - M1 modèle du système réel
  - M2 métamodèle
  - M3 métamétamodèle (M3) auto-descriptif

Exemple

Les niveaux du MOF



source : Illustration of the Meta-Object Facility.

Modélisation à quatre niveaux

	OMG	Lang. de prog.	XML
M0	instances	exécution	données réelles
M1	diag. de classes	prog.	données en XML
M2	MM UML	grammaire	schéma XML
M3	MOF	EBNF	XML

Chapitre IX : Notions de métamodélisation

29 Métaformalisme
- Introduction
- MOF
- EMF

Eclipse Modeling Framework

- Le projet Eclipse EMF est une variante du MOF pour Eclipse
- Ce projet fournit une infrastructure de génération de code en se basant sur des modèles
- A partir de modèles exprimés avec XMI, il est par exemple possible de générer des classes Java
- Le niveau M3 de EMF est nommé *ECore*

Chapitre IX : Notions de métamodélisation

Chapitre IX : Notions de métamodélisation

UML et le MOF

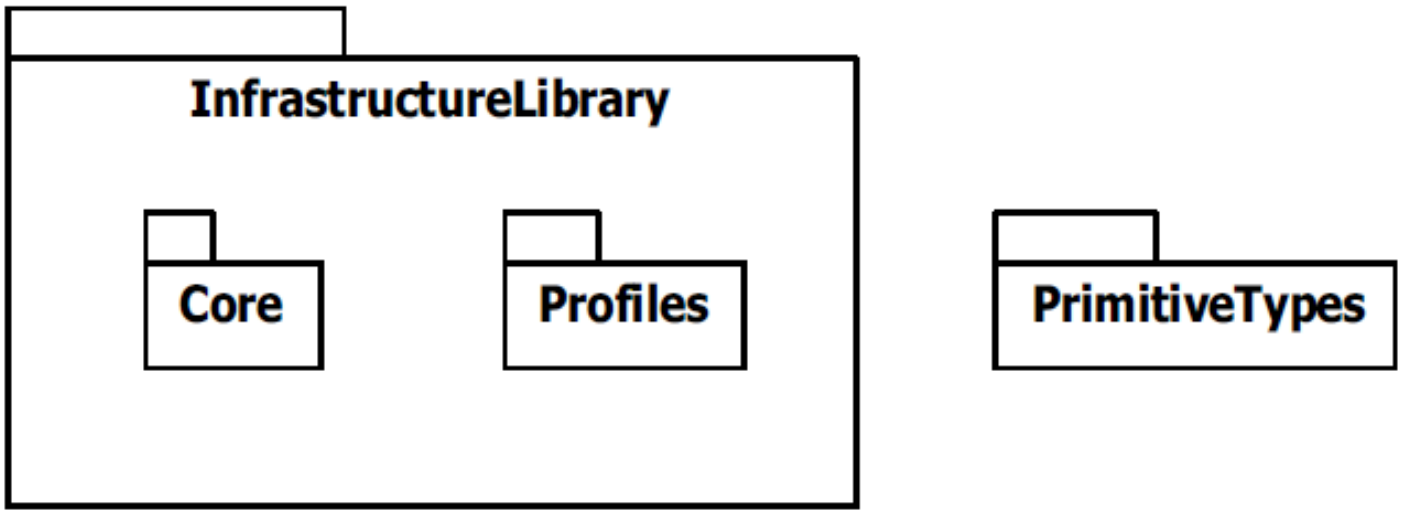
- MOF 2 est le métaformalisme de l'OMG pour la description de métamodèles
- UML 2 est le métamodèle dédié à la modélisation d'applications OO et basé sur le MOF
- Un sous-ensemble du diagramme de classe UML est utilisée pour définir le MOF
- La spécification *UML 2 Infrastructure* regroupe les définitions communes à UML et au MOF
- La spécification *UML 2 Superstructure* réutilise certaines parties de *UML 2 Infrastructure*

UML 2 Infrastructure

- La spécification UML 2 Infrastructure définit un ensemble de packages
- À la racine, on trouve les packages *InfrastructureLibrary* et *PrimitiveTypes*
  - *InfrastructureLibrary* définit le métalangage (package *Core*) et des méchanismes d'extensions (package *Profile*)
  - *PrimitiveTypes* regroupe des types de bases
- Les packages peuvent être réutilisés grâce à l'opération de *merge*

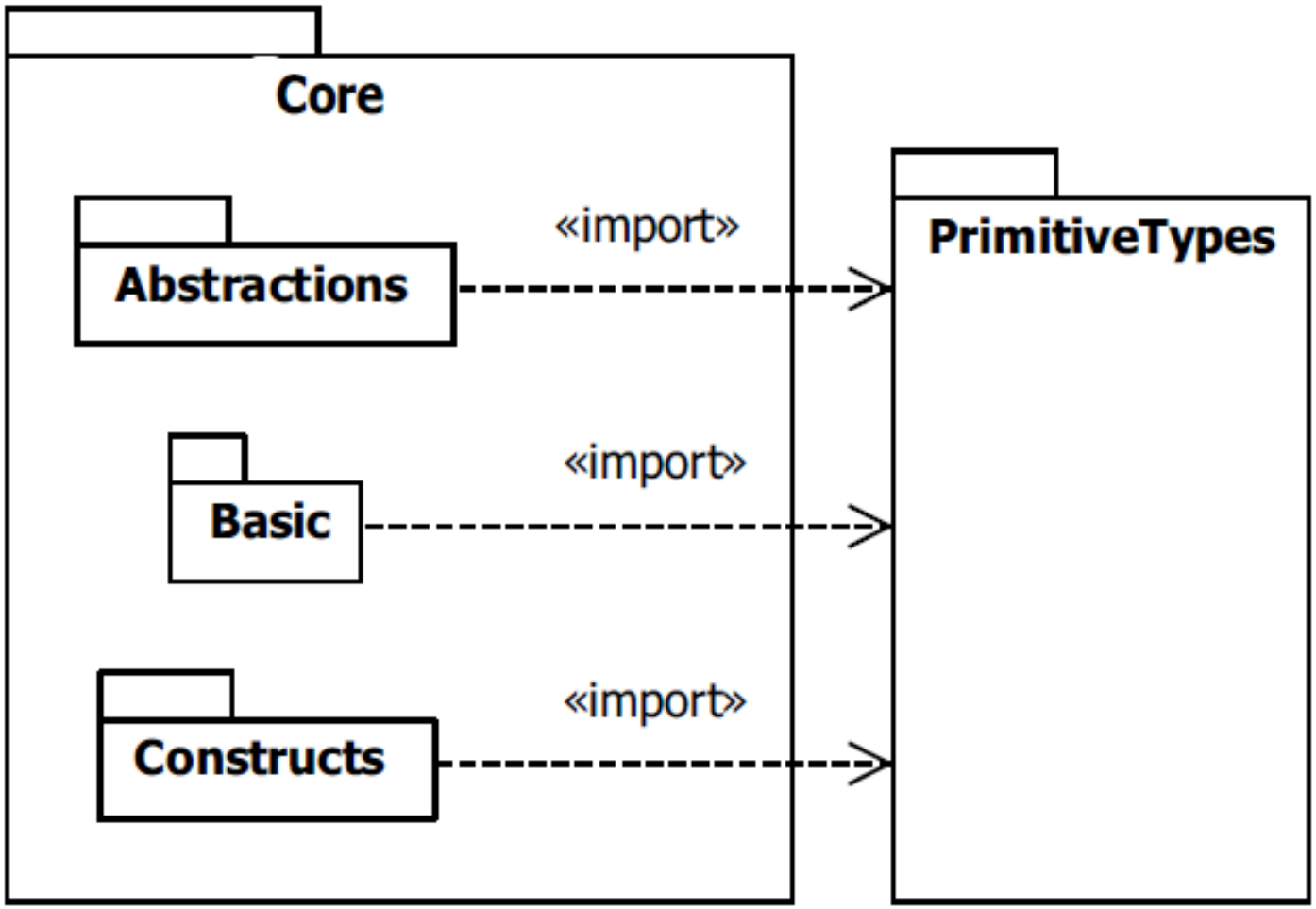
Exemple

UML 2 Infrastructure



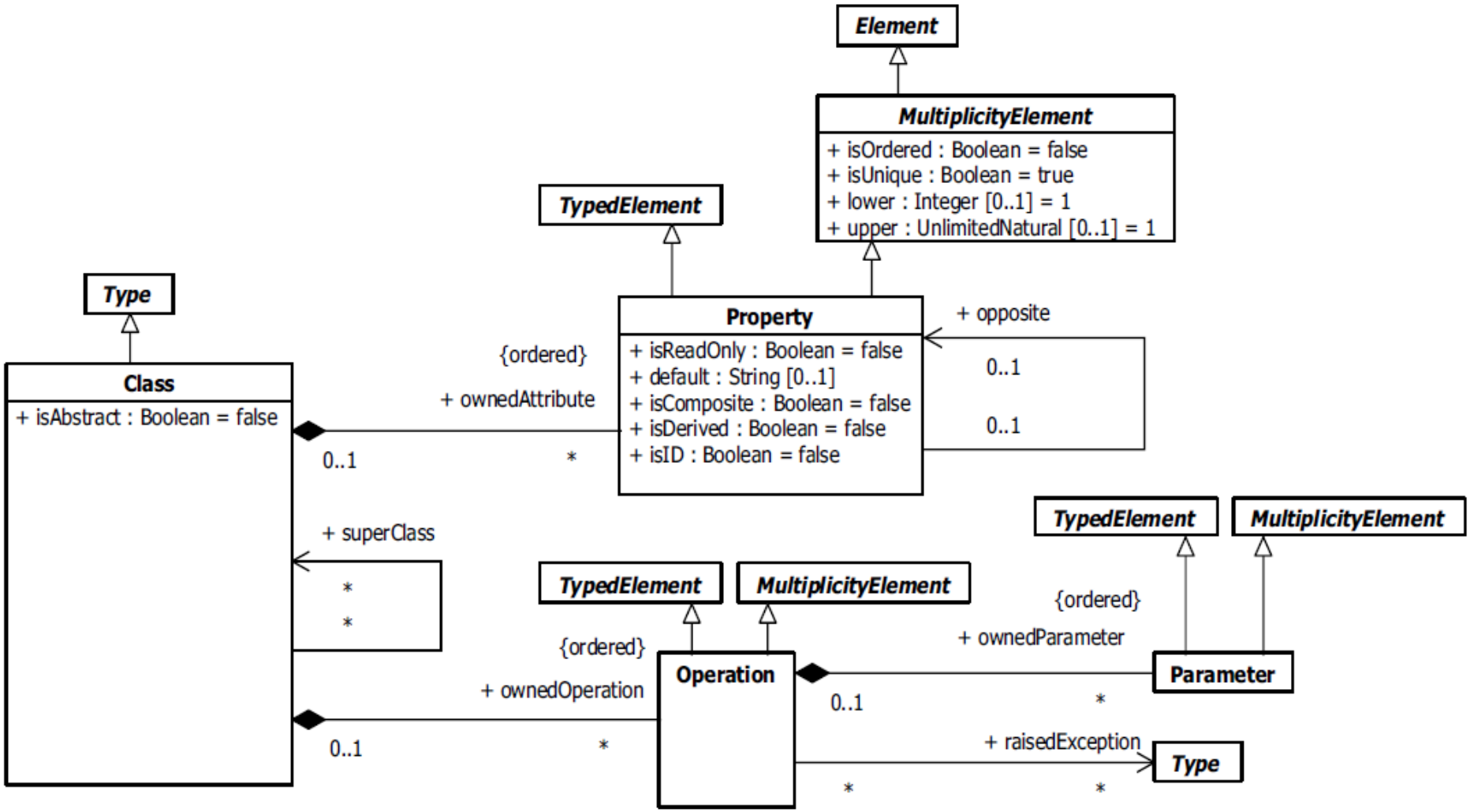
Exemple

Le package *Core*



Exemple

Diagramme de classes défini dans *Core*::*Basic*

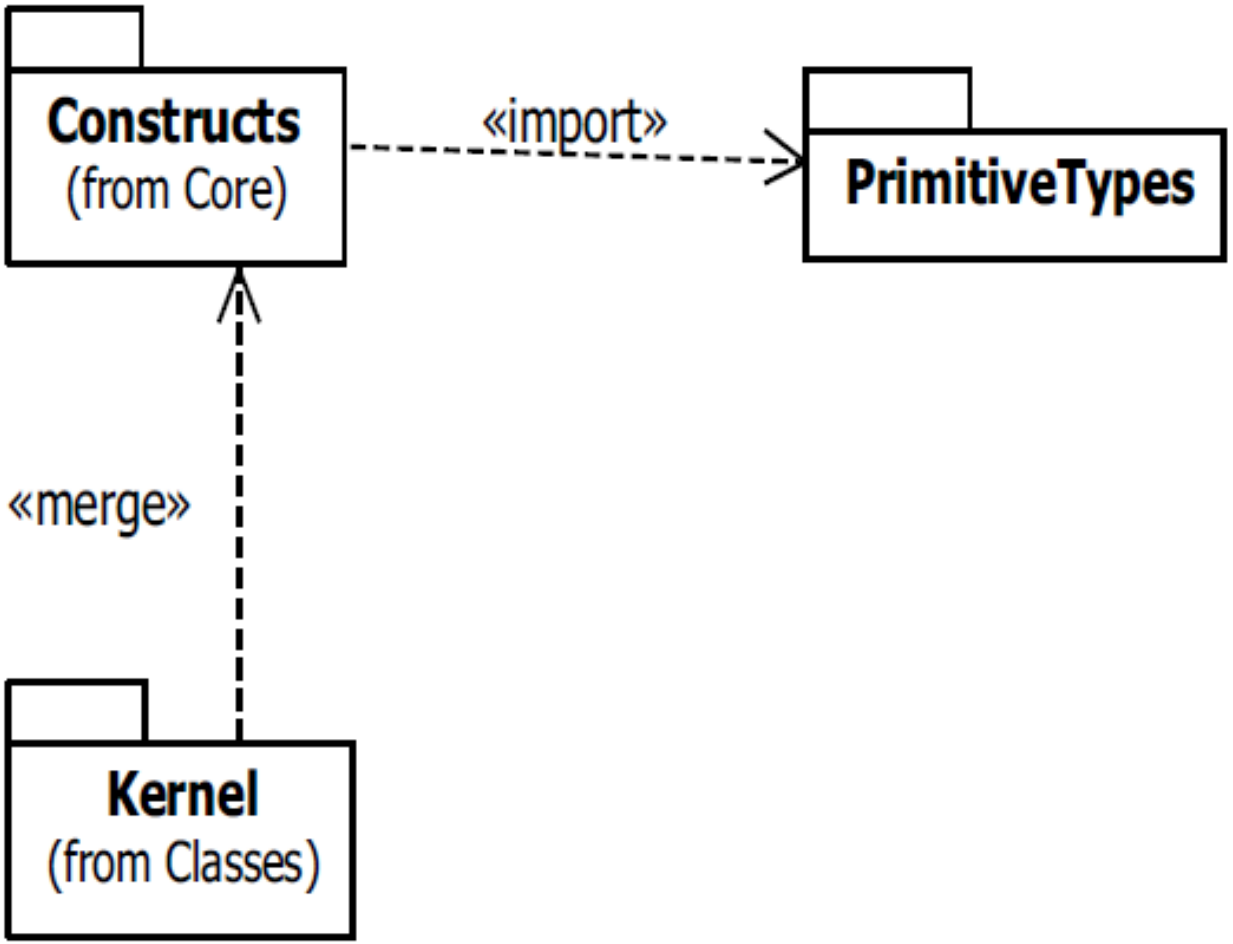


UML 2 Superstructure

- UML 2 Superstructure définit les concepts nécessaires à l'ensemble des diagrammes UML
- Il utilise l'opération de *merge* pour importer des packages de UML 2 Infrastructure
- Il est découpé en de nombreux packages (*Classes*, *Components*, *Deployments*, ...)

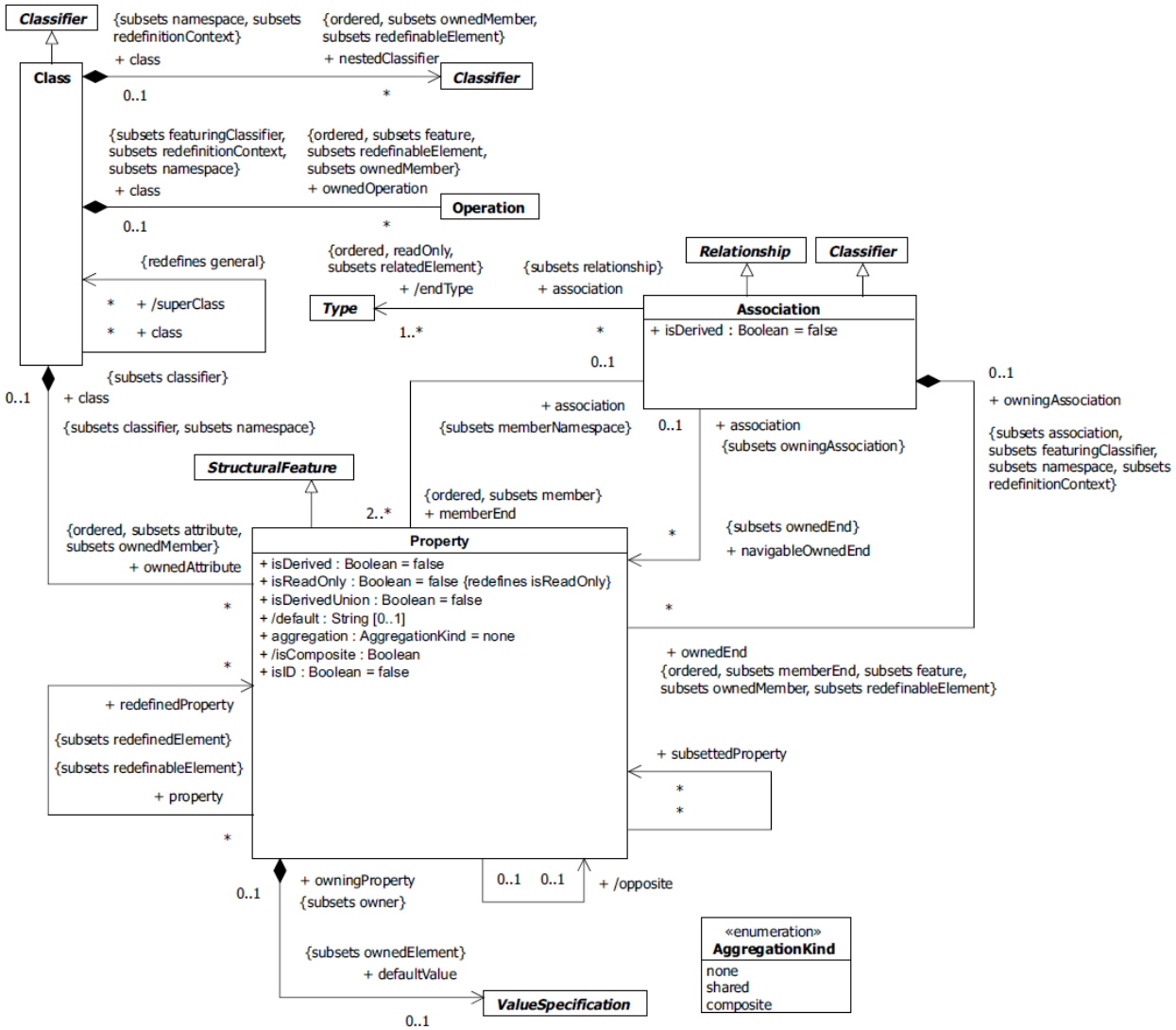
Exemple

Packages de UML 2 Infrastructure fusionné dans *Classes*::*Kernel*



Exemple

Diagramme de classes dans *Classes*::*Kernel*



MOF 2

- MOF 2 a été décomposée en deux parties
  - EMOF *Essential MOF* décrit des métamodèles sans association
  - CMOF *Complete MOF* décrit des métamodèles avec association
- EMOF intègre le package *Core*::*Basic* de UML 2 Infrastructure
- CMOF intègre le package *Core*::*Construct* de UML 2 Infrastructure

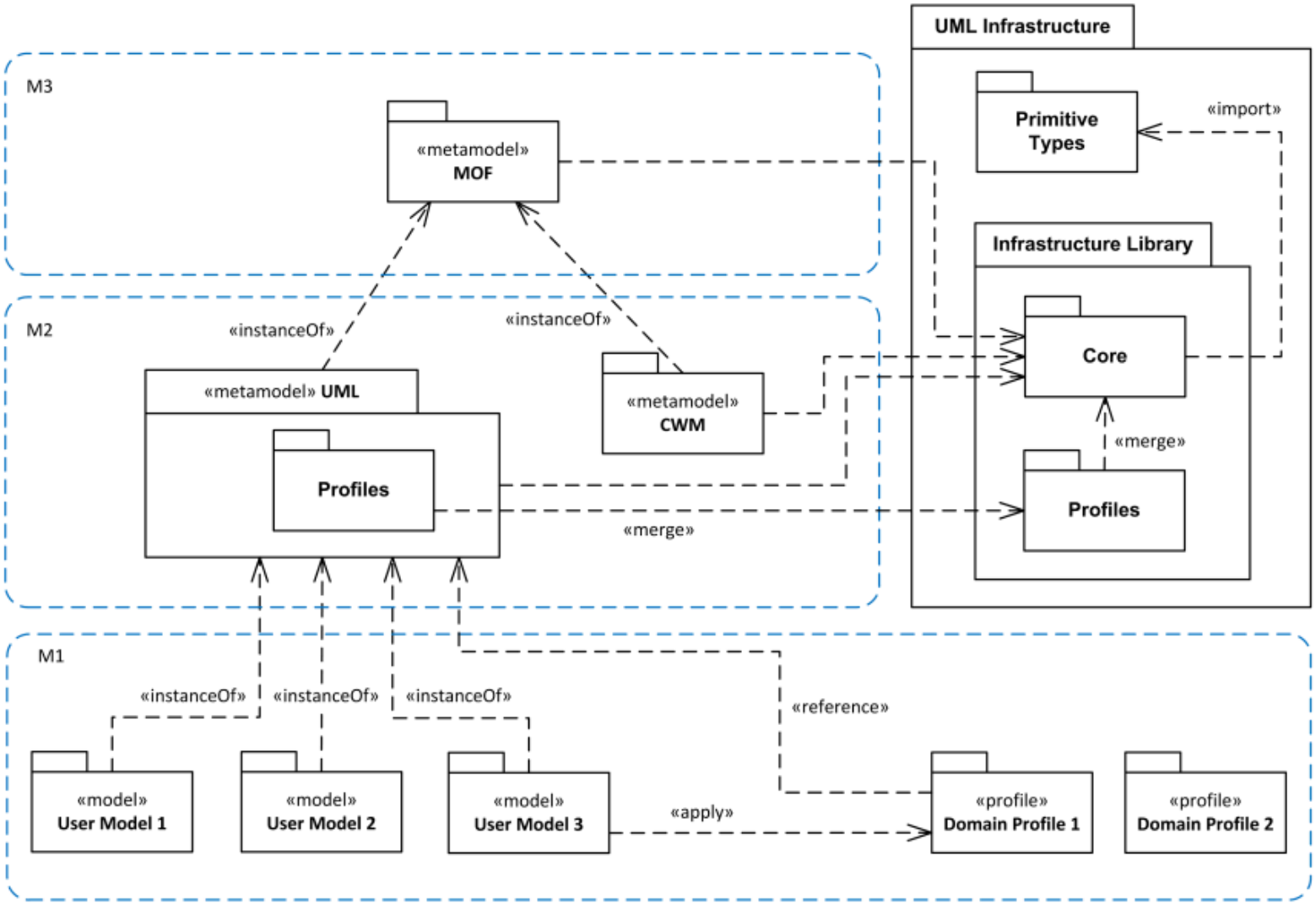
Chapitre IX : Notions de métamodélisation

30 Métaformalisme et UML
- Métamodèle d'UML et du MOF
- Étendre UML

Profil UML

- Les concepts génériques d'UML peuvent être appliqués dans de nombreux domaines (application OO, métamodèle, schéma de BD, ...)
- Le problème est alors de déterminer le sens du modèle
- Un *profil UML* permet de préciser le sens de ces modèles
- Il permet d'adapter dynamiquement (sans changer le métamodèle) UML à un domaine particulier
- Un profil UML définit un ensemble de stéréotypes qui seront ensuite utilisés pour préciser la signaification de concepts UML

Vue d'ensemble



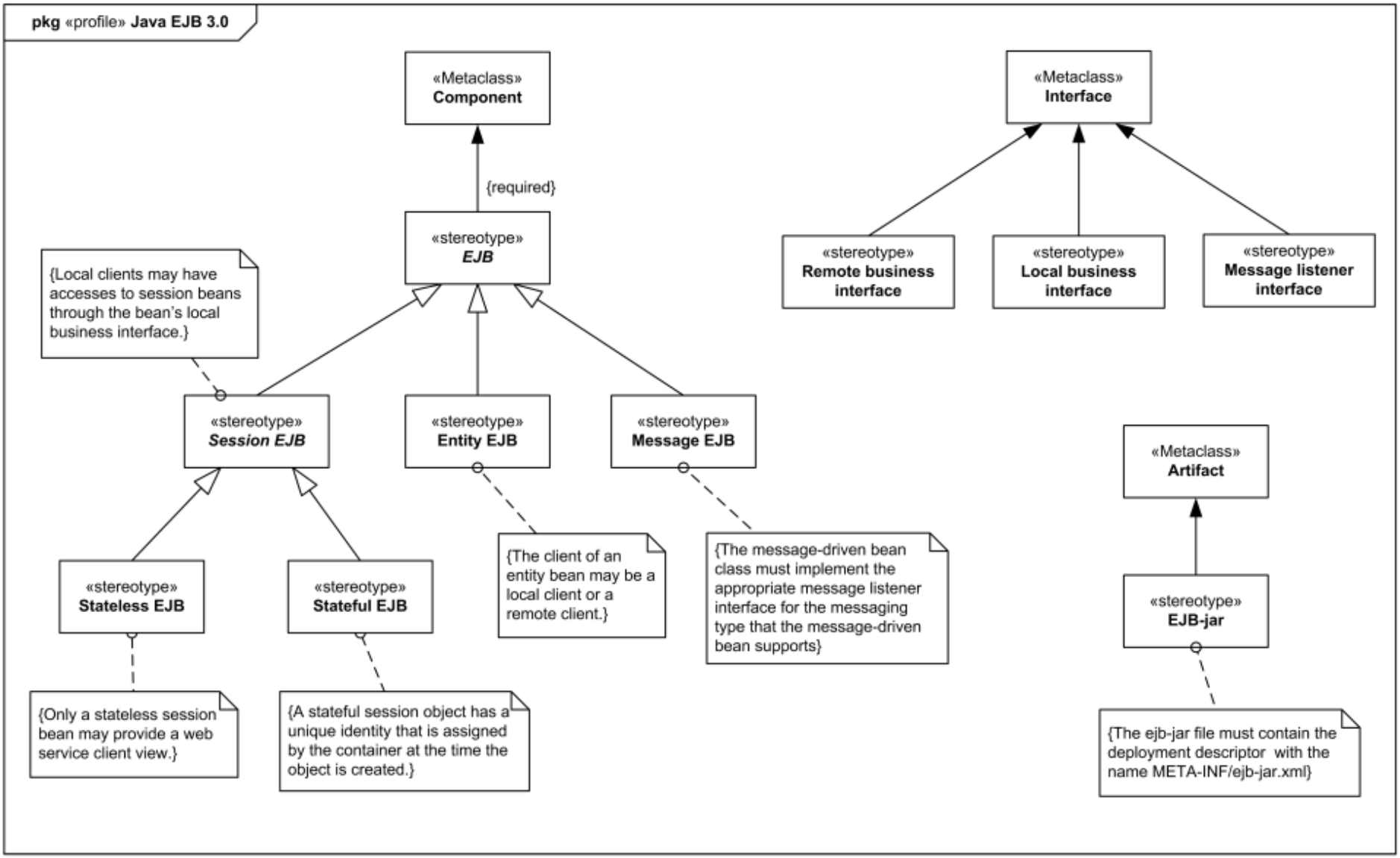
source : Meta Meta Models, UML and Profiles.

Diagramme de profil

- Un diagramme de profil est un diagramme de structure permettant de définir des stéréotypes, des mots-clés ou des contraintes
- Un profil utilise la notation des packages avec le mot-clé « `profile` »
- Il peut uniquement introduire des stéréotypes, des mots-clés ou des contraintes sur des métaclasses existantes
- Un stéréotype est représenté comme une classe avec le stéréotype « `stereotype` »
- Le lien d'extension entre un stéréotype et une métaclasse est représenté par une flèche continue terminée par un triangle plein (*association extension*)
- Un stéréotype peut comporter des propriétés (mots-clé)

Exemple

Un profil UML pour EJB 3



Quelques profils existants

SysML *System Modeling Language* pour l'ingénierie de système

CWM *Common Warehouse Metamodel* pour l'échange de métadonnées BI

UTP *UML testing profile* pour les tests

Chapitre IX : Notions de métamodélisation

Introduction

- L'*ingénierie dirigée par les modèles* (*Model-Driven Engineering* ou *MDE*) est une approche pour le développement logiciel s'appuyant sur les modèles et leurs transformations
- L'approche *Model-Driven Architecture* (MDA) est la proposition de l'OMG dans ce cadre
- MDA inclut plusieurs standards comme UML, MOF et XMI
- Les objectifs de MDA sont
  - élaborer des modèles perennes,
  - indépendants des détails techniques,
  - permettre la génération automatique du code,
  - obtenir un gain de productivité.
- MDA repose sur trois modèles

  CIM le modèle des exigences
  PIM le modèle d'analyse et de conception
  PSM le modèle de code

CIM

- Le modèle des exigences (*Computation Independant Model* ou *CIM*) spécifie les exigences du client
- A partir de ces besoins, un lien de traçabilité est maintenu tout au long du processus
- Ce modèle ne comprend pas d'information sur la réalisation de l'application ou les traitements

PIM

- Le modèle d'analyse et de conception (*Platform Independant Model* ou *PIM*)
- Ce modèle décrit les modules et sous-modules composant l'application
- Il demeure indépendant de toute plateforme d'implémentation (JavaEE, .NET, PHP, ...)

PSM

- Le modèle de code (*Platform Specific Model* ou *PSM*) décrit l'implémentation d'une application sur une plateforme technique
- Il a pour objectif de faciliter la génération de code
- Il s'appuie sur les profils UML pour les plateformes.

Transformation de modèles

- MDA utilise la transformation de modèles pour dériver le PIM à partir du CIM et le PSM à partir du PIM
- Ces transformations sont également modélisées dans MDA