



Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 1:

Software and Software Engineering

1.1 La Nature du Logiciel...

Le logiciel est intangible

- Il est difficile de gérer l'effort de développement

Le logiciel est facile à reproduire

- Tout le coût se trouve dans son développement
 - Pour d'autres produits, la fabrication est souvent le processus le plus coûteux

L'industrie du logiciel exige beaucoup de main d'œuvre

- Le processus de développement est difficile à automatiser



La nature du logiciel

Même des informaticiens peu qualifiés peuvent arriver à bricoler quelque chose qui semble fonctionner

- Les problèmes de qualité d'un logiciel ne sont pas apparents

Un logiciel est facile à modifier

- La tentation est forte d'effectuer des changements rapides sans vraiment en mesurer la portée

Un logiciel ne s'use pas

- Il se détériore à mesure que des changements sont effectués
 - en raison de l'introduction d'erreurs
 - ou par une complexification indue



La nature du logiciel

Conclusions

- Beaucoup de logiciels **sont mal conçus et se détériore rapidement**
- L'ingénierie du logiciel est une nécessité



Les différentes catégories de logiciel...

Système embarqué en temps réel

- Exemple: contrôle et surveillance des systèmes
- Doit réagir rapidement
- Sûreté est essentielle

Logiciel de traitement de données

- Utilisé pour diriger les entreprises
- Précision et sécurité sont essentielles

Logiciel de jeu

Logiciel d'appareil mobile

Logiciel de Web

Etc.

1.2 Qu'est-ce que le génie du logiciel?...

Le processus visant la **résolution de problèmes posés par un client** par le développement systématique et l'évolution de systèmes logiciels de grande taille et de haute qualité en respectant les contraintes de coûts , de temps, et autres.



Qu'est-ce que le génie du logiciel?...

...la résolution de problèmes posés par un client...

- Voilà le **but** essentiel du génie logiciel
- Dans certains cas, la solution peut être de ne **rien développer**, si un produit satisfaisant existe déjà
- L'ajout de fonctionnalités inutiles réduit souvent la qualité du logiciel
- L'ingénieur logiciel doit établir *une bonne communication* afin de bien identifier et comprendre le problème à résoudre



Qu'est-ce que le génie du logiciel?...

...par le **développement systématique** et l'évolution...

- Tout processus d'ingénierie implique l'application de techniques bien maîtrisées de façon organisée et disciplinée
- Plusieurs pratiques reconnues ont maintenant été standardisées
 - e.g. IEEE ou ISO
- La plupart des projets logiciels consiste à faire évoluer un logiciel existant



Qu'est-ce que le génie du logiciel?...

...systèmes logiciels de grande taille et de haute qualité...

- Un logiciel de grande taille est un logiciel *qui ne peut être compris* par une seule personne
- Le travail en équipe et une bonne coordination sont essentiels
- Défi principal: subdiviser le travail à accomplir tout en s'assurant que chacune des parties fonctionneront harmonieusement ensemble
- Le produit final doit rencontrer des critères de qualité bien établis



Qu'est-ce que le génie du logiciel?...

...en respectant les contraintes de coûts , de temps, et autres.

- Les ressources sont limitées
- Le bénéfice résultant doit être supérieur aux coûts
- D'autres entreprises se font concurrence pour effectuer le travail à moindre coût et plus rapidement
- Une mauvaise estimation des coûts et de la durée du projet peut mener à l'échec du projet

1.3 La profession d'ingénieur logiciel

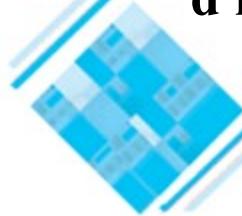
Le terme Génie Logiciel a été introduit en 1968

- Il s'agissait de reconnaître le fait que les principes du génie peuvent s'appliquer au développement du logiciel

Le génie est une pratique régulée par une corporation professionnel

- Protection du public
- Les ingénieurs conçoivent des artefacts selon des pratiques bien acceptées qui impliquent l'application de la science, des mathématiques et de l'économie
- Pratiques conformes à une éthique établie

Dans de nombreux pays, une grande partie du génie logiciel ne nécessite pas de licence d'ingénierie, mais il s'agit toujours d'ingénierie



Code d'éthique du génie logiciel

Les ingénieurs en logiciel doivent

- Agir conformément à l'intérêt public
- Agir au mieux des intérêts de leurs clients
- Développer et maintenir avec les normes les plus élevées possibles
- Maintenir l'intégrité et l'indépendance
- Promouvoir une approche éthique en gestion
- Faire progresser l'intégrité et la réputation de la profession
- Soyez juste et solidaire envers vos collègues
- Participez à l'apprentissage tout au long de la vie

1.4 Les parties impliquées dans le génie du logiciel

1. Utilisateurs

- Ceux qui se servent du logiciel

2. Clients

- Ceux qui paient pour le logiciel

3. Développeurs

- Ceux qui conçoivent le logiciel

4. Gestionnaires

- Ceux qui supervisent la production du logiciel

Tous ces rôle peuvent être remplis par la même personne



1.5 La qualité du logiciel...

Convivialité

- Apprentissage ais , facilit  d'utilisation

Efficacit 

- Aucun gaspillage de ressources (m moire, temps de calcul, ...)

Fiabilit 

- Les t ches sont effectu s sans probl mes

Facilit  de maintenance

- Ais  à modifier, à faire ´voluer

R utilisabilit 

- Ses parties peuvent  tre r utilis s facilement



La qualité du logiciel...

Client:

Résoud le problème
à un coût acceptable

Utilisateur:

Facile à apprendre,
utile et efficace

Perception de
La qualité du
logiciel

Développeur:

Facile à concevoir,
à maintenir, à réutiliser

Gestionnaire:

Se vend bien,
satisfait les clients,
peu coûteux à développer

La qualité du logiciel...

Ces différents attributs peuvent être en conflit

- Accroître l'efficacité peut rendre le logiciel plus difficile à maintenir et à réutiliser

Définir des critères de qualité constitue un élément clé du génie du logiciel

- La conception a alors pour objectif de rencontrer ces critères
- Trop en faire est une perte de temps et de ressources



1.6 Projets de génie logiciel

La plupart des projets consiste à faire évoluer ou à maintenir un logiciel existant dont on a hérité de la responsabilité

- Projets correctifs: corriger des défauts
- Projets adaptatifs: modifications à apporter au système de façon à tenir compte de changement dans
 - Le système d'opération
 - Les données ou la base de données
 - Les règles et procédures
- Projets d'amélioration: ajout de nouvelles options
- Projets perfectifs: changements apportés à la structure interne du programme

Projets de génie logiciel

Développement à partir de zéro

- Concevoir un nouveau produit
- Il s'agit là de la minorité des projets entrepris



1.7 Activités communes aux projets de génie logiciel

Définition et spécification des exigences

- Ce qui inclut
 - Analyse de domaine
 - Définition du problème
 - Collection des exigences
 - Obtenir des contributions du plus grand nombre de sources possible
 - Analyse des exigences
 - Organiser l'information
 - Spécification formelle des exigences
 - Rédaction d'instructions détaillées sur la façon dont le logiciel doit se comporter

Activités communes aux projets de génie logiciel

Conception

- Décider comment la technologie disponible sera utilisée pour répondre aux besoins
- Ce qui inclut:
 - Déterminer ce qui sera réalisé par le logiciel et par le matériel
 - Mettre au point l'architecture du système, la définition des sous-systèmes et de leurs interactions
 - Élaboration des éléments internes de chaque sous-système
 - Conception des interfaces usagers et des bases de données



Activités communes aux projets de génie logiciel

Modélisation

- Créer des représentations du logiciel et de son domaine d'application
 - Modélisation de son utilisation (cas d'utilisation)
 - Modélisation de sa structure
 - Modélisation de sa dynamique et de son comportement

Programmation

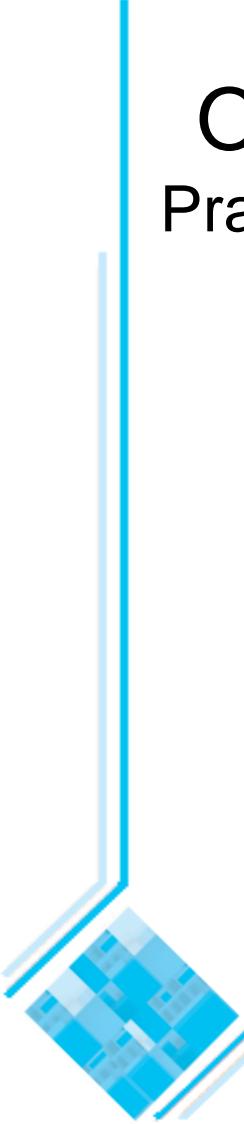
Assurance de qualité

- Révision et inspections
- Mise à l'épreuve

Déploiement

Gestion du processus





Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 2A:

Review of Object Orientation

2.1 Qu'est-ce que l'Orientation Objet?

Paradigme procédural:

- Le logiciel est organisé autour de la notion de *procédures*
- *Abstraction de procédures*
 - Fonctionne bien lorsque les données sont simples

Abstraction de données

- Grouper ensemble les données décrivant une même entité
- Aide à réduire la complexité du système

Paradigme orienté objet:

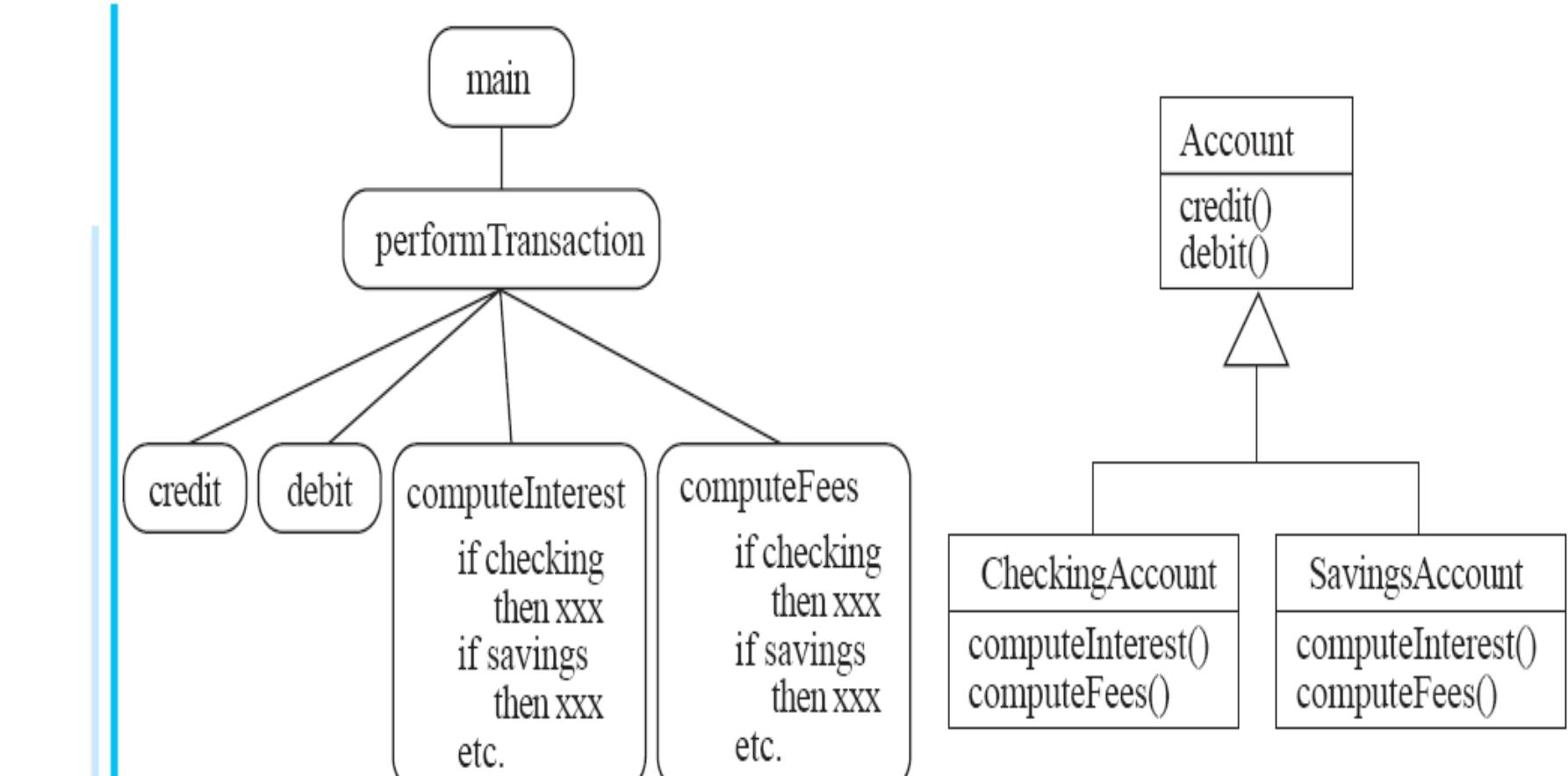
- Les abstractions de procédures sont placées à l'intérieur des abstractions de données

Paradigme Orienté Objet

Une approche consistant à organiser la solution d'un problème autour du concept d'objets.

- Ces objets sont des instances de classes:
 - Ce sont des abstractions de données
 - Contenant des abstractions de procédures
- Un programme devient alors un ensemble d'objets collaborant entre eux afin d'effectuer une tâche donnée

Illustration des ces deux paradigmes

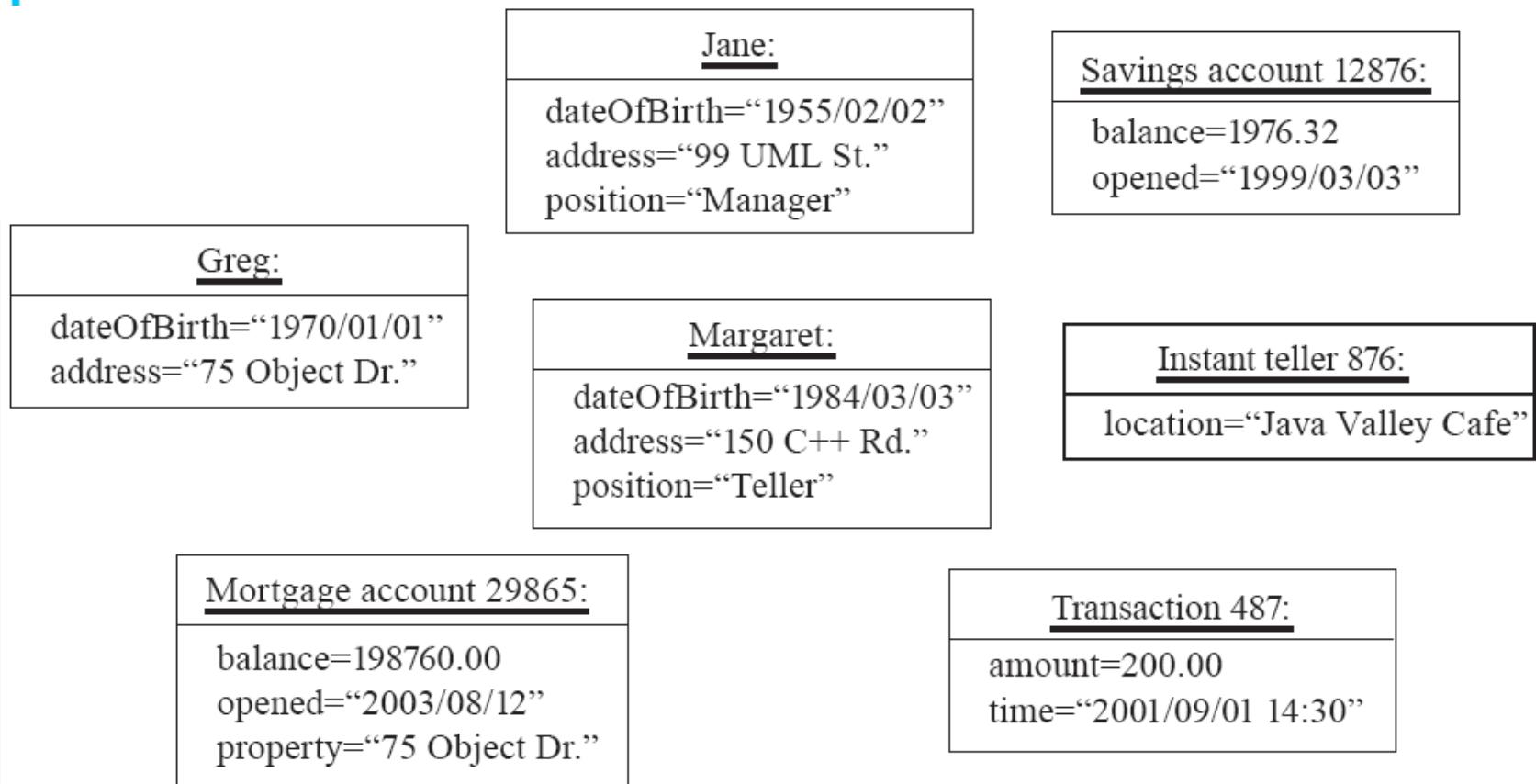


2.2 Classes et Objets

Un objet

- est un ensemble structuré de données s'exécutant dans un logiciel
- a des *propriétés*
 - représentant son état
- a un *comportement*
 - définissant ses actions et réactions
 - simulant parfois le comportement d'un objet du monde réel

Objets



Classes

Une classe:

- Est une unité d'abstraction dans un programme orienté-objet
- Représente des objets similaires
 - ses *instances*
- Est un module logiciel
 - Décrivant la structure de ses instances (propriétés)
 - Contenant des *méthodes* définissant leur comportement

Classe ou instance?

- Quelque chose pouvant avoir des instances est une classe
- Quelque chose est une *instance* si il est l'un des éléments d'un ensemble (défini par la classe)

Film

- Classe; ses instances sont les différents films.

Bobine de Film:

- Classe; ses instances sont chacune des bobines existantes

Bobine de film portant le numéro de série SW19876

- Une instance de **BobineDeFilm**

Science Fiction

- Instance de la classe **Genre**.

Film de Science Fiction

- Classe; une de ses instances est le film ‘Star Wars’

Représentation du film ‘Star Wars’ au cinéma Le Phoenix à 19h:

- Une instance de **Representation**

Donner un nom à une classe

- Devrait toujours débuter par une lettre majuscule
 - E.g. Banque **not** banque
- Utiliser le *singulier*
- Utiliser le bon niveau de généralité
 - E.g. Municipalité pourrait être préférable à Ville
- Donner un nom non ambigu ayant un sens précis
 - E.g. Pièce peut avoir plusieurs sens

2.3 Variables d'instances

Ce sont des variables définies à l'intérieur d'une classe

- Attributs
 - des données simples
 - E.g. nom, dateDeNaissance
- Associations
 - représentent une relation avec d'autres classes
 - E.g. superviseur, coursSuivis
 - Plus d'explications à venir au Chapitre 5

Variables vs. Objets

Une variable

- *Réfère* à un objet
- Peut référer à différents objets à différents instants

Un objet peut être simultanément référé par plus d'une variable

Type d'une variable

- Détermine quelle classe d'objets elle peut référer

Variables de classe

La valeur d'une telle variable est partagée par toutes les instances de la classe

- Aussi appelée *variable statique*
- Si l'une des instances modifie la valeur d'une variable de classe, alors toutes les autres instances verront ce changement
- Ces variables sont utiles pour:
 - représenter des constantes (e.g. PI)
 - représenter des propriétés appliquant à une classe en général

Attention: *ne pas faire un usage excessif des variables de classes*

2.4 Méthodes, Opérations et Polymorphisme

Opération

- Une abstraction procédurale de haut niveau correspondant à un comportement spécifique
- Indépendante de toute implémentation
 - E.g., calcul de l'aire d'une forme géométrique



Méthodes, Opérations et Polymorphisme

Méthode

- Une abstraction de procédure utilisée pour implémenter un comportement dans une classe donnée
- Plusieurs classes différentes peuvent avoir des méthodes de même nom
 - Généralement c'est qu'elles réalisent la même opération d'une manière propre à chaque classe
 - E.g, le calcul de l'aire d'un rectangle et d'un cercle

Polymorphisme

Une propriété importante liée au paradigme orienté-objet selon laquelle une même opération s'applique de différente façon dans différentes classes

- Exige l'existence de plusieurs méthodes ayant le même nom
- La méthode qui sera exécutée par un objet dépend de la classe d'appartenance de cet objet
- Réduit grandement la nécessité d'insérer des énoncés de contrôle tels que le `if-else` ou le `switch`

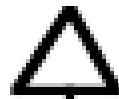
2.5 Organisation des Classes en Hiérarchies

Super-classe

- contient les éléments communs à un ensemble de sous-classes

Arbre d'héritage

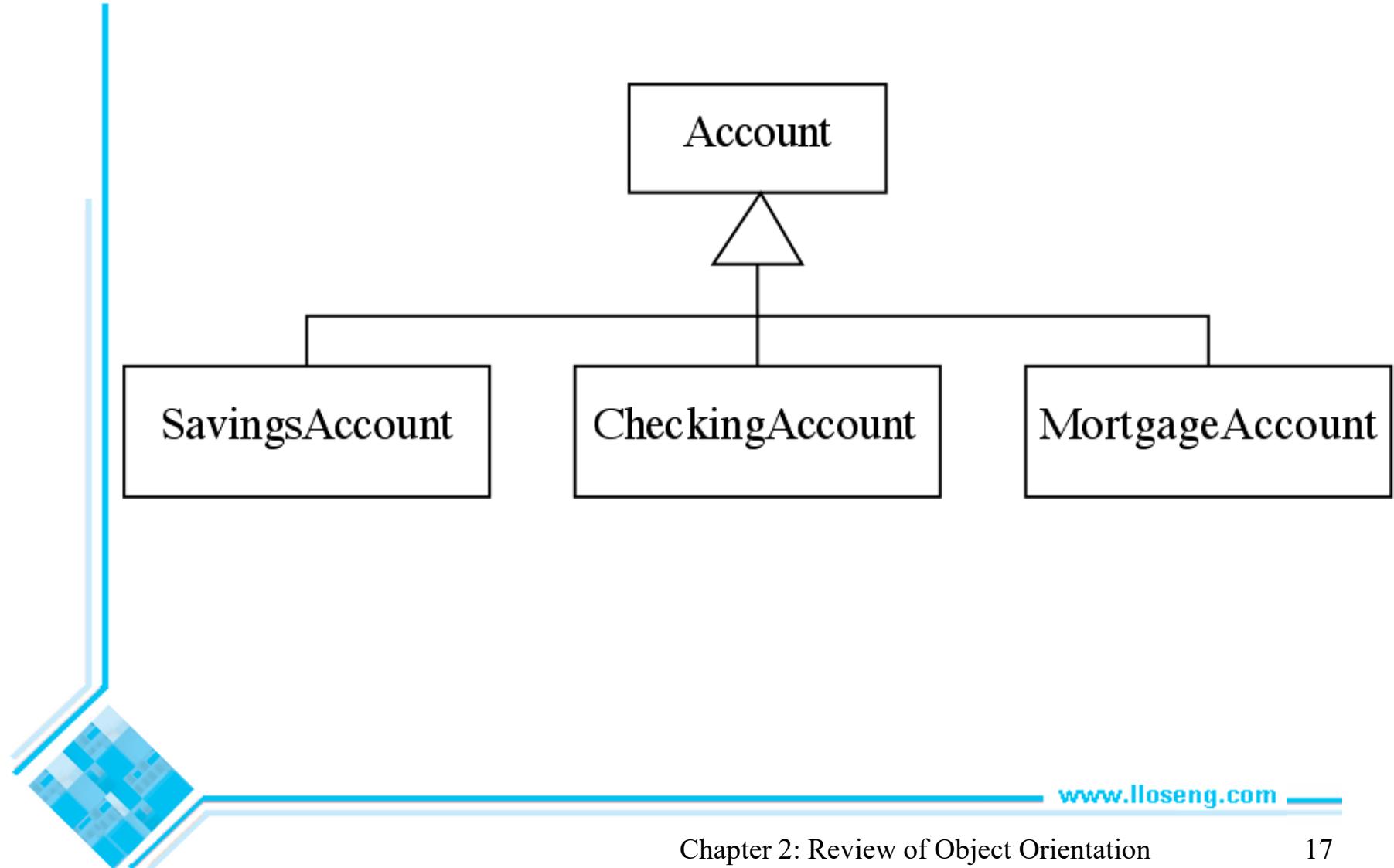
- Montre la relation entre super-classes et sous-classes
- Le triangle est le symbole utilisé pour représenter une *généralisation*



Héritage

- Le mécanisme selon lequel toutes les sous-classes possèdent *implicitement* les éléments de leurs super-classes

Un exemple d'arbre d'héritage



Règle “est-un(e)”

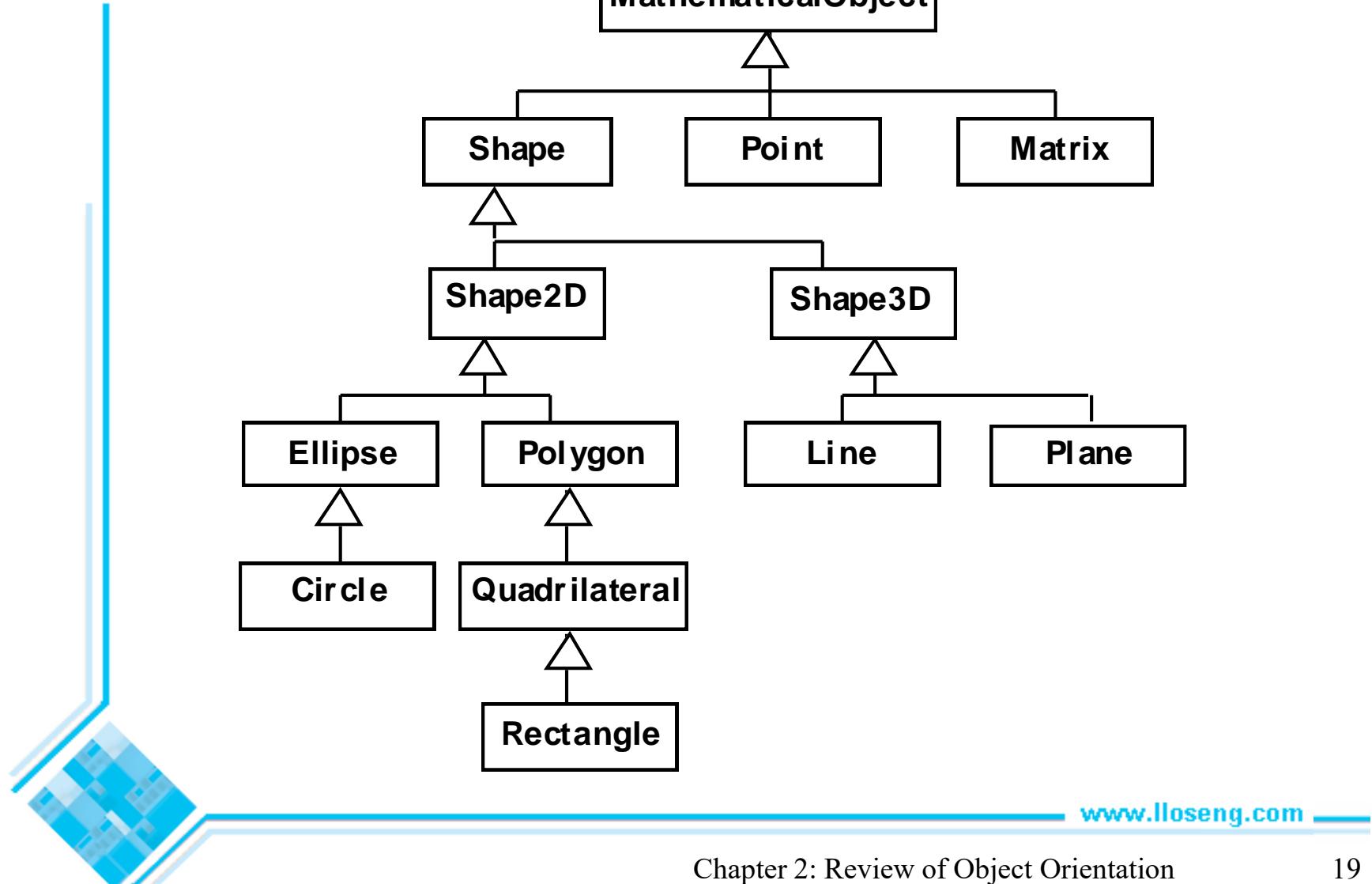
La généralisation obéit toujours à la règle « est-un(e) »

- “Un compte chèque *est un* compte de banque”
- “Un village *est une* municipalité”

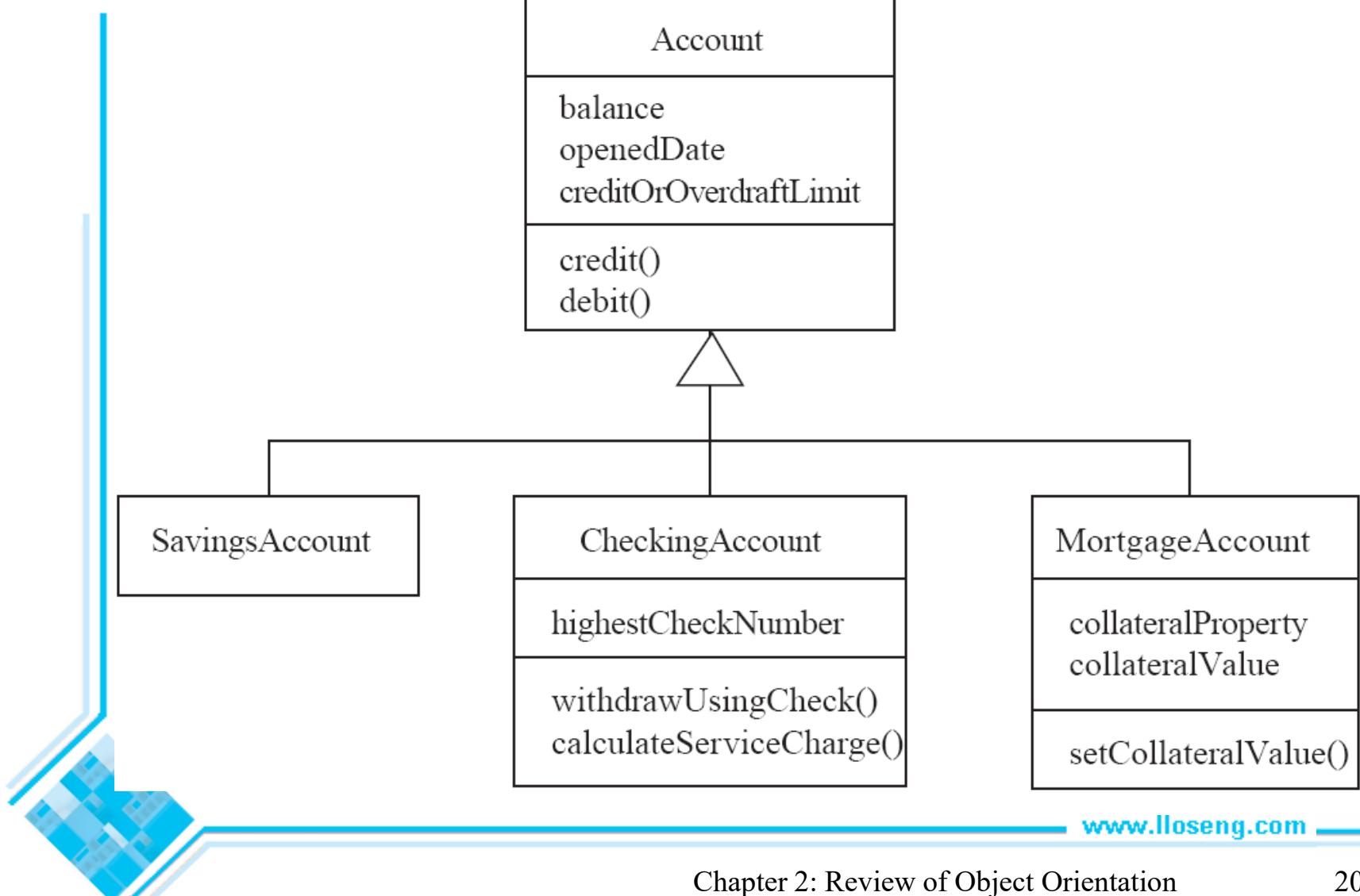
Est-ce qu’une Province devrait être une sous-classe de Pays?

- Non!
 - “Une province *n’est pas un* pays”

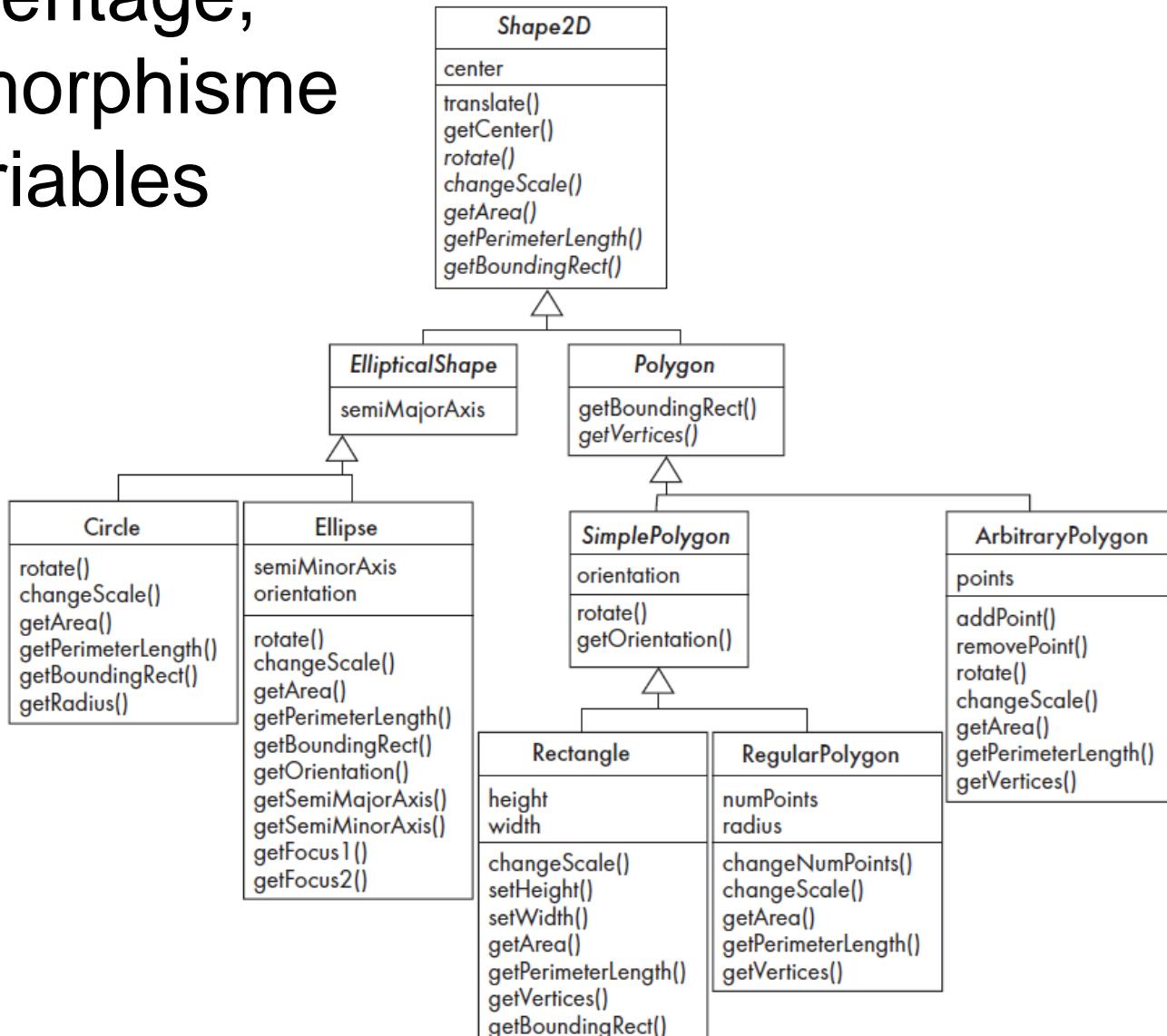
Un arbre d'héritage pour des entités géométriques



Tous les éléments hérités doivent s'appliquer adéquatement aux sous-classes

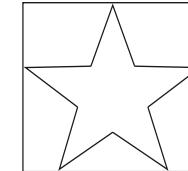
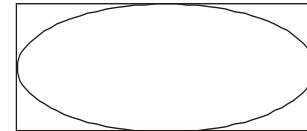


2.6 Héritage, polymorphisme et variables

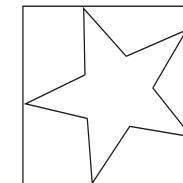
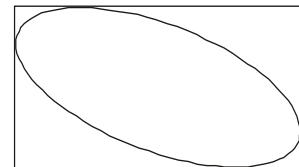


Quelques opérations pour des formes géométriques

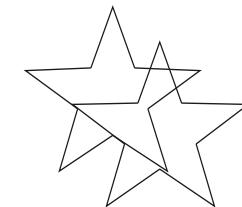
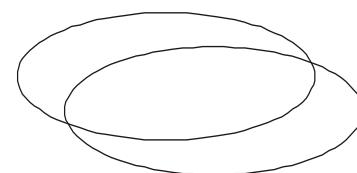
Original objects
(showing bounding rectangle)



Rotated objects
(showing bounding rectangle)



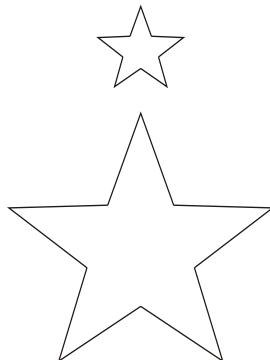
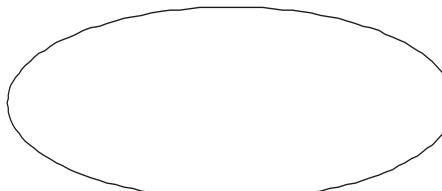
Translated objects
(showing original)



Scaled objects
(50%)



Scaled objects
(150%)



Classes abstraites et Méthodes abstraites

Une opération doit être définie au plus haut niveau d'abstraction possible

- A ce niveau, l'opération peut être *abstraite*
- Dans ce cas la classe devient aussi *abstraite*
 - Aucun instance directe peut être créée
 - Une classe non abstraite est une classe concrète
- Si une classe possède une opération abstraite, l'une de ses sous-classe doit définir concrètement cette opération
 - Toutes les opérations d'une classe au bas de l'arbre (feuille) d'héritage doivent être concrètes

Redéfinition

Une méthode héritée peut être redéfinie

- Dans un but de restriction
 - E.g. `scale(x, y)` ne fonctionne pas dans `Circle`
- Dans un but d'extension
 - E.g. `SavingsAccount` peut inclure des frais additionnels dans le cas d'un retrait
- Dans un but d'optimization
 - E.g. la méthode `getPerimeterLength` dans `Circle` est beaucoup plus simple que celle de `Ellipse`

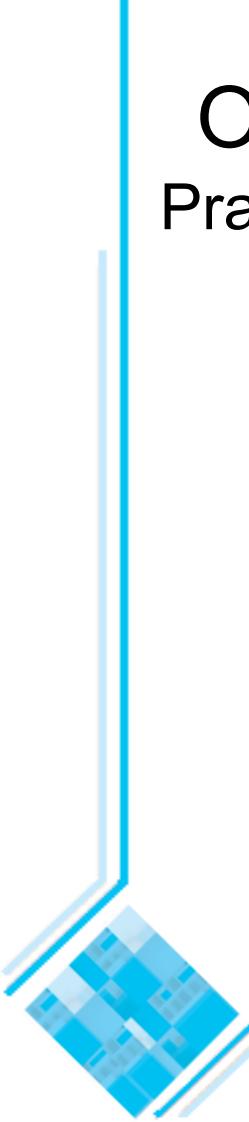
Comment se détermine la méthode qui sera exécutée

1. D'abord vérifier si il existe une définition pour cette méthode dans la classe de l'objet de référence
2. Sinon, vérifier dans la super-classe de niveau immédiatement supérieur
3. Répéter l'étape 2, en remontant les niveaux jusqu'à ce qu'une méthode concrète soit trouvée
4. Si aucune méthode concrète n'est trouvée, il y a erreur
 - En Java et C++ une telle erreur est identifiée à la compilation

Liaison dynamique

Se produit lorsque la décision concernant la méthode à exécuter se prend lors de l'exécution du programme

- Nécessaire lors que:
 - Une variable réfère à un objet dont la classe d'appartenance fait partie d'une hiérarchie
 - Et lorsque plus d'une méthode existe pour une même opération (polymorphique)



Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 2B:

Review of Java

Le langage Java

Histoire

- Le premier langage orienté objet fut Simula-67
 - Conçu pour faciliter l'écriture de programmes de simulation
- Au début des années 1980, Smalltalk fut développé à Xerox PARC
 - Syntaxe nouvelle, importante librairie à code ouvert, indépendance de plate-forme, ramasse-miette, bytecode
- A la fin des années 1980, C++ fut développé par B. Stroustrup,
 - Tire profit des avantages de l'orienté-objet tout en profitant de la popularité de C
- En 1991, les ingénieurs de Sun Microsystems lance un projet afin concevoir un langage à être utilisé dans les petits appareils intelligents: Oak
 - Avec l'avènement de l'Internet, une nouvelle opportunité se dessine pour cette technologie
 - Ce nouveau langage renommé Java, fut officiellement lancé en 1995 à la conférence SunWorld '95

Documentation Java

La recherche de classes et de méthodes est une compétence essentielle

- La recherche de classes et de méthodes inconnues vous permettra de comprendre le code

La documentation Java peut être générée automatiquement par un programme appelé Javadoc

- La documentation est générée à partir du code et de ses commentaires
- Vous devez formater vos commentaires comme indiqué dans certains des exemples du livre
 - Ceux-ci peuvent inclure du HTML intégré

Documentation Java

```
/*
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute <a href="#{@link}">{@link URL}</a>. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

Revue de Java

Les prochaines diapositives vous rappelleront de plusieurs fonctionnalités clés de Java

- Pas dans le livre
- Consultez le site Web du livre pour
 - Un aperçu plus détaillé de Java
 - Pointeurs vers des tutoriels, des livres, etc.



Caractères et chaînes

Character est une classe représentant des caractères Unicode

- Plus d'un octet chacun
- Représenter n'importe quelle langue du monde

char est un type de données primitif contenant un caractère Unicode

String est une classe contenant des collections de caractères

- + est l'opérateur utilisé pour concaténer des chaînes

Tableaux et collections

Les tableaux sont de taille fixe et manquent de méthodes pour les manipuler

ArrayList est la classe la plus largement utilisée pour contenir une *collection* d'objets

- Plus puissant que les tableaux, mais moins efficace

Les *itérateurs* sont utilisés pour accéder aux membres des *listes*

```
a = new ArrayList();
Iterator i = a.iterator();
while(i.hasNext())
{
    aMethod(i.next());
}
```

Casting

Java est très strict sur les types

- Casting pour élargissement - conversion d'un type plus petit en un type plus grand
 - byte -> short
- Casting for reduction - conversion d'un type plus grand en un type de taille plus petite
 - long -> int



Exceptions

Tout ce qui peut mal aller devrait potentiellement générer une exception

- **L'exception** est une classe avec de nombreuses sous-classes pour des choses spécifiques qui peuvent mal aller

Utilisez un bloc try - catch pour intercepter une exception

```
try
{
    // some code
}
catch (ArithmeticException e)
{
    // code to handle division by zero
}
```

Interfaces

Comme les classes abstraites, mais ne peuvent pas avoir d'instructions exécutables

- Définir un ensemble d'opérations qui font du sens dans plusieurs classes
- Types de données abstraites

Une classe peut implémenter n'importe quel nombre d'interfaces

- Il doit avoir des méthodes concrètes pour les opérations

Vous pouvez déclarer le type d'une variable d'être une interface

- C'est comme si vous déclarez le type d'être une classe abstraite

Les interfaces importantes de la librairie Java incluent

- Runnable, Collection, Iterator, Comparable, Cloneable

Packages et importation

Un package combine des classes associées dans des sous-systèmes

- Toutes les classes dans un répertoire particulier

Les classes de différents packages peuvent avoir le même nom

- Bien que non recommandé

L'importation d'un package se fait comme suit:

```
import finance.banking.accounts.*;
```

Contrôle d'accès

S'applique aux méthodes et aux variables

- Public
 - Toute classe peut accéder
- Protected
 - Seul le code du package ou des sous-classes peuvent accéder
- (Vide)
 - Seul le code du package peut accéder
- Private
 - Seul le code écrit dans la classe peut accéder
 - L'héritage se produit toujours!

Threads et concurrence

Thread:

- Séquence d'exécution des instructions pouvant être exécutées simultanément avec d'autres threads

Pour créer un thread en Java:

1. Créez une classe implémentant Runnable ou étendant Thread
2. Implémentez la méthode `run` comme une boucle qui fait quelque chose pendant un certain temps
3. Créez une instance de cette classe
4. Appelez l'opération `start`, qui appelle `run`

Directives de style de programmation

Toujours garder à l'esprit qu'un programme est fait pour être lu

- Choisissez toujours l'alternative la plus simple
- Rejeter le code intelligent mais difficile à comprendre
- Un code plus court n'est pas nécessairement meilleur

Choisissez les bons noms

- Assurez-vous qu'ils sont hautement descriptifs
- Ne vous inquiétez pas d'utiliser des noms longs

Style de programmation...

Commentez extensivement

- Commentez tout ce qui n'est pas évident
- Ne commentez pas ce qui est évident
- Les commentaires doivent représenter 25 à 50% du code

Organiser les éléments de classe de manière cohérente

- Variables, constructeurs, méthodes publiques puis méthodes privées

Structurer le code de façon consistante

Style de programmation...

Évitez la duplication de code

- Ne pas « cloner » si possible
 - Créez une nouvelle méthode et appelez-la
 - Le clonage donne deux copies qui peuvent toutes deux avoir des bogues
 - Lorsqu'une copie du bogue est corrigée, l'autre peut être oubliée



Style de programmation ...

Adhérez à de bons principes orientés objet

- Par exemple, la règle « est-un »

Préférez le privé au public

Ne mélangez pas le code d'interface usager avec le restant du code

- Interagir avec l'utilisateur dans des classes séparées
 - Cela rend les classes non-UI plus réutilisables



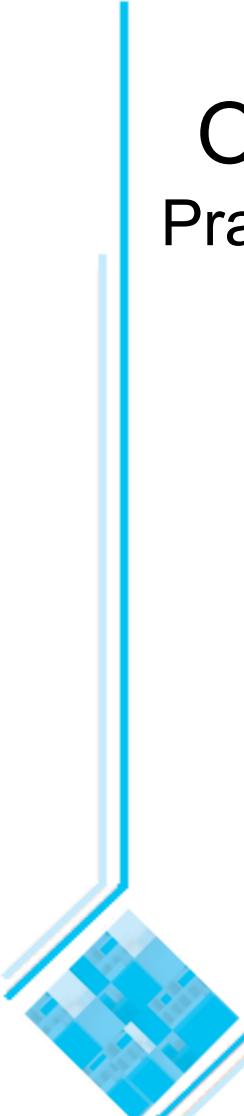
Difficultés et risques de programmation

Évolution du langage et fonctionnalités obsolètes:

- Java évolue, certaines fonctionnalités deviennent « obsolètes » à chaque version

L'efficacité peut être un problème dans certains systèmes orientés objet

- Java peut être moins efficace que d'autres langages
 - Basé sur VM
 - Liaison dynamique



Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 3A:

Basing Software Development on Reusable Technology

3.1 Profiter de l'expérience des autres

Tout bon ingénieur logiciel devrait éviter de développer à nouveau du logiciel qui a déjà été développé par d'autres

Il doit plutôt chercher à réutiliser:

- Réutiliser **l'expertise** des autres
- Réutiliser des algorithmes ou des **designs standards**
- Réutiliser des **librairies** de classes ou de procédures
- Réutiliser des **commandes** disponibles dans le langages ou le système d'exploitation utilisé
- Réutiliser des **cadres d'applications**
- Réutiliser **des applications**

3.3 Cadriel: des sous-systèmes réutilisables

Un *cadriel* ou *cadre d'application (framework)* est un logiciel réutilisable qui propose une solution générique à un problème généralisable.

- Il fournit les services que requiert différentes applications.

Principe: Plusieurs applications réalisant des tâches différentes bien que semblables tendent à avoir un design similaire

Les cadres d'application pour promouvoir la réutilisation

Un cadre d'application est intrinsèquement *incomplet*

- Certaine classes ou méthodes utilisées par le cadre sont manquantes (*ouvertures*, “*slots*”)
- Certaines fonctionnalités sont optionnels
 - Celles-ci sont mises à la disposition du développeur (*crochets*, “*hooks*”)
- Les développeurs utilisent les *services* qu'offrent le cadre d'application
 - L'ensemble des service s'appelle le *API* (Application Program Interface)

Cadre d'application orienté objet

Conformément au paradigme orienté objet, un cadre d'application sera composé d'un ensemble de classes.

- Le API est alors l'ensemble de toutes les **méthodes publiques** de ces classes.
- Quelques unes de ces classes seront abstraites

Exemples:

- Pour la gestion d'un service de paiement
- Pour un club d'achat, de points
- Pour un système d'inscription à des cours
- Pour des sites de commerce électronique

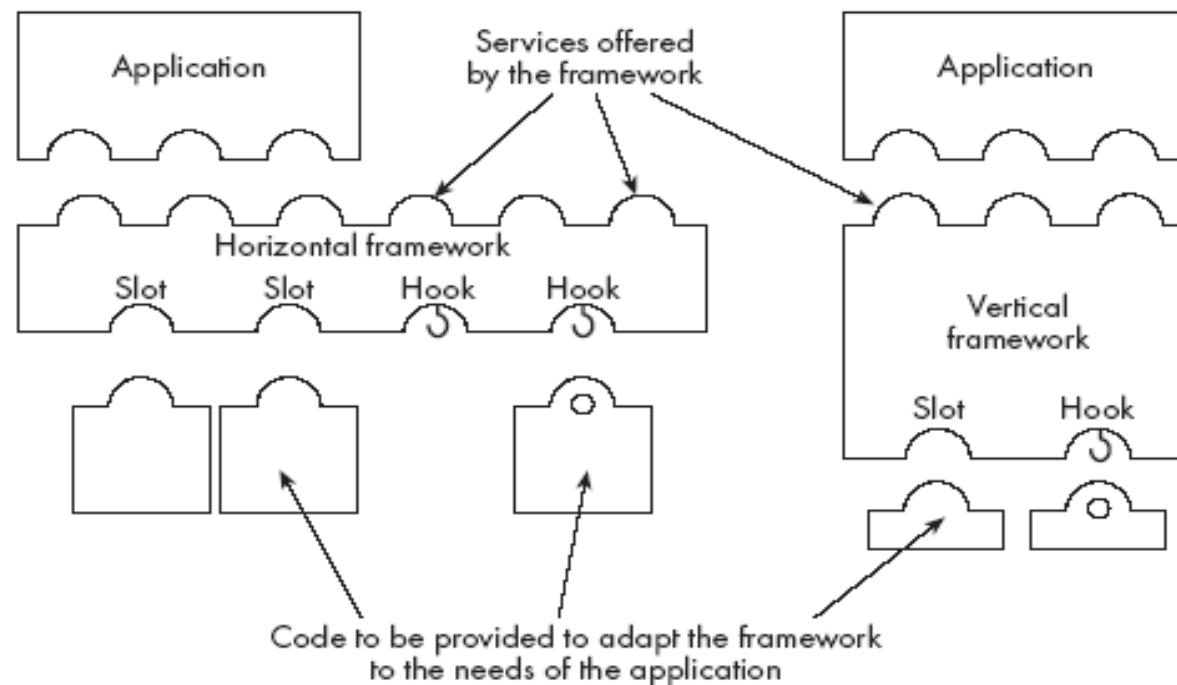
Cadriels et ligne de produits

Une ligne de produit (ou famille de produit) est un ensemble de produits construits sur une base technologique commune

- Les différents produits possèdent **différentes caractéristiques** afin de satisfaire différents marchés
- La technologie **commune** est incluse dans le cadriel
- Chaque produit complète les **ouvertures et crochets disponibles**
 - Exemple: les version *demo, de base, pro* d'un logiciel

Types de cadres d'applications

- Un cadre *horizontal* fournit des services généraux qu'un grand nombre d'application peuvent utiliser
- Un cadre *vertical* est beaucoup plus complet, seules demeurent quelques ouvertures qui doivent être définies afin de s'adapter à une application spécifique



3.4 L'architecture Client-Server

Un système *distribué* est un système dans lequel:

- les calculs sont effectués par des programmes distincts
- et peuvent s'exécuter sur des machines différentes
- Coopérant ensemble dans la réalisation d'une tâche

Serveur:

- C'est un programme qui fournit des services à d'autres programmes se connectant à celui-ci par l'intermédiaire d'un canal de communication

Client:

- C'est un programme qui accède à un serveur afin d'obtenir des services
- Plusieurs clients peuvent se connecter à un serveur simultanément

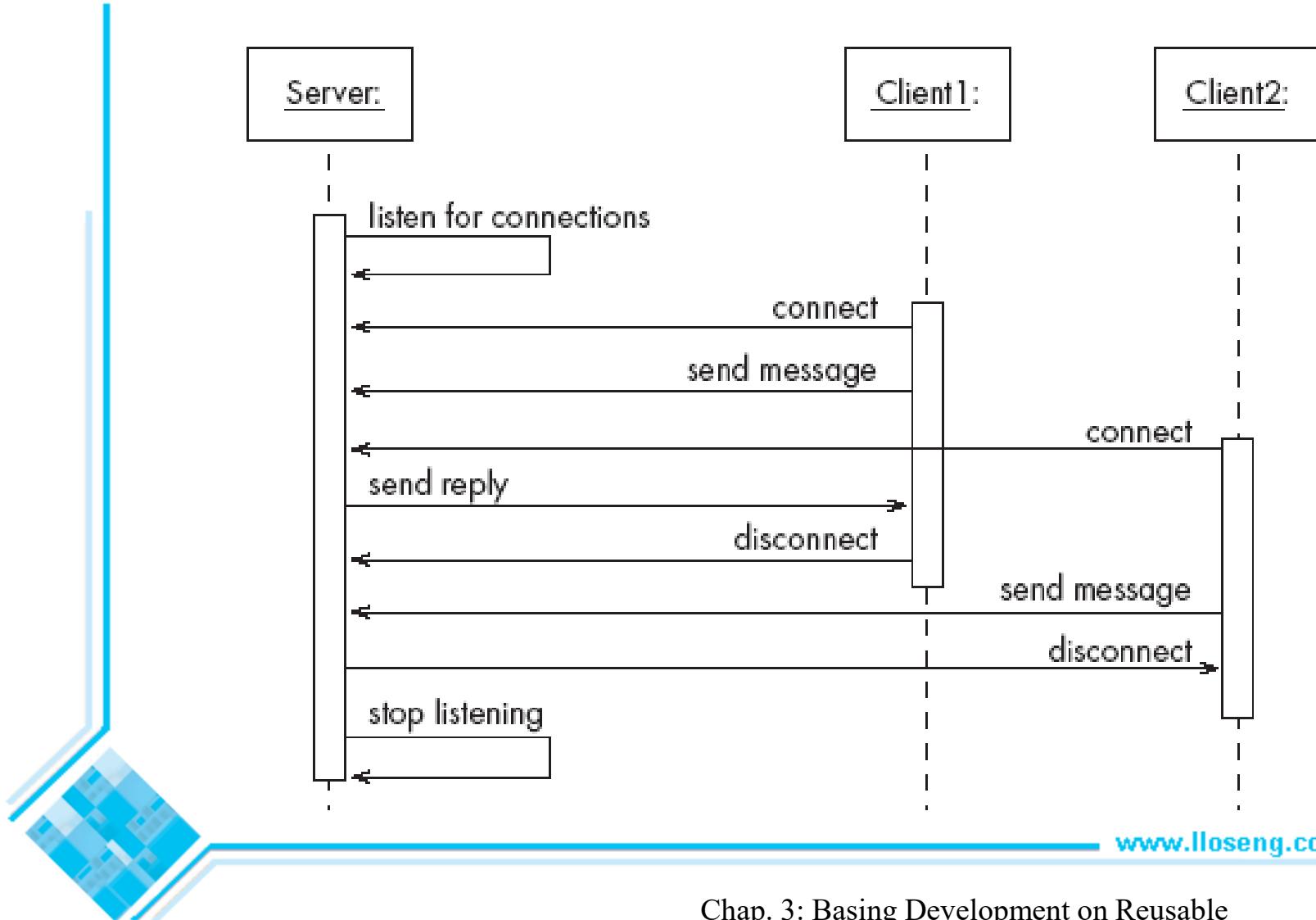
Exemples de systèmes client-serveur

- Le World Wide Web
- Le courrier électronique
- Les système de fichiers distribués (*NFS*)
- Les systèmes de transaction à distance
- Les systèmes d'affichage à distance
- Les systèmes de communication
- Les systèmes de bases de données distribuées

Séquence d'activités dans un système client-server

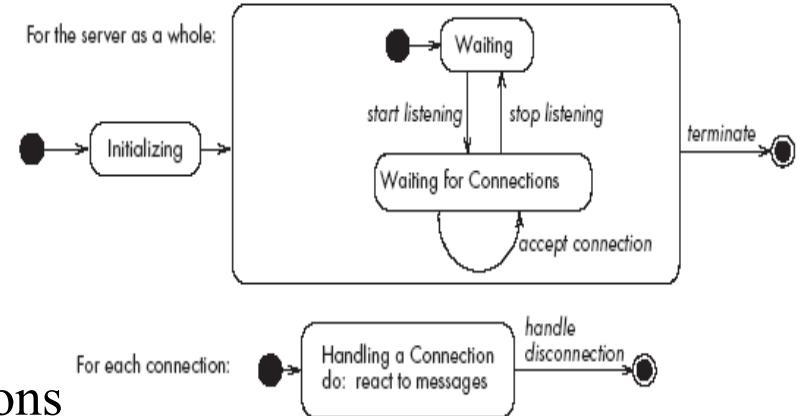
1. Le serveur débute son exécution
2. Le serveur se met en attente de connexions
3. Un client débute son exécution et effectue une série d'opérations
 - Certaines de ces opérations impliquent l'accès à certains des services qu'offre le serveur
4. Le client tente alors de se connecter au serveur, celui-ci accepte ou refuse la connexion
5. Le serveur attend de recevoir des messages en provenance des clients connectés
6. Lorsqu'un message est reçu, le serveur effectue certaines opérations puis se remet en attente
7. Clients et serveur répète ce manège jusqu'à ce que l'un de ceux-ci se déconnecte ou soit stoppé.

Diagramme montrant un serveur communiquant avec deux clients



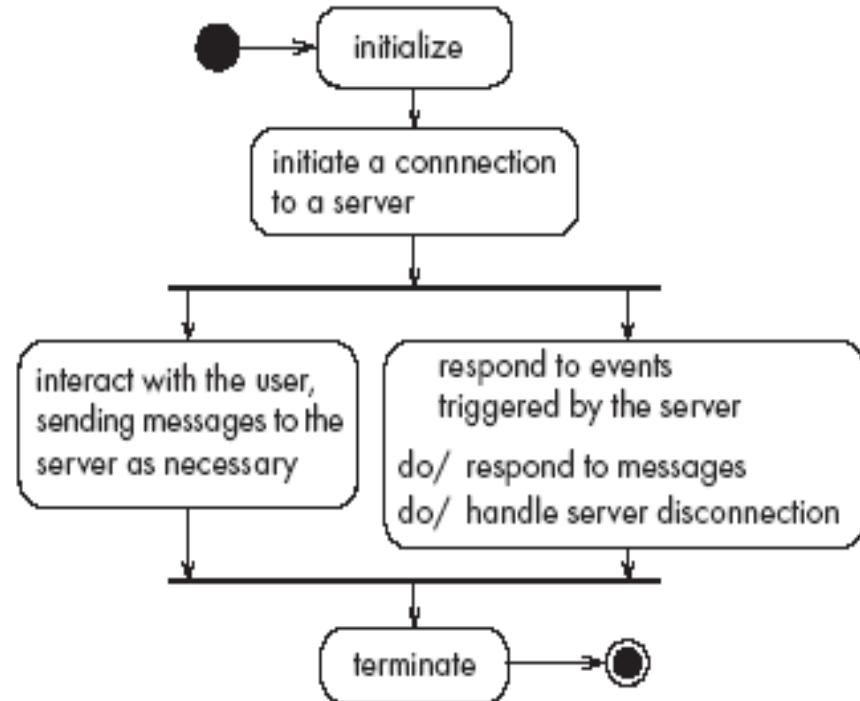
Activités effectuées par un serveur

1. Initialisation
2. Mise en attente de clients
3. Gestion des événements suivants:
 1. Accepter de nouvelles connexions
 2. Répondre aux messages
 3. Permettre la déconnexion de clients
4. Arrêt l'attente
5. Terminaison

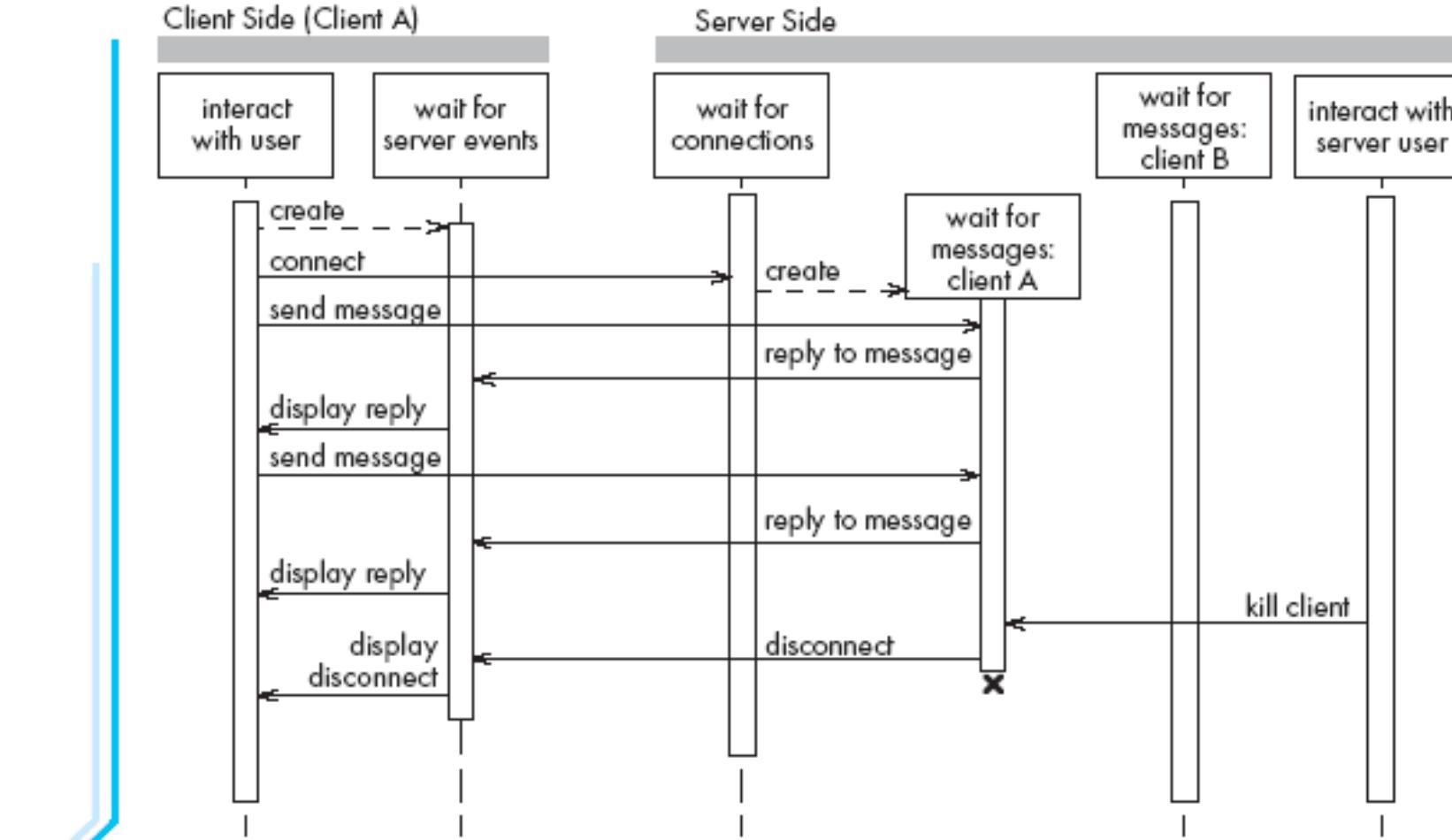


Activités effectuées par un client

1. Initialisation
2. Connexion au serveur
3. Envoie de messages
4. Gestion des événements suivants:
 1. Répondre aux messages
 2. Permettre la déconnexion du serveur
5. Terminaison



Fils d'exécution d'un système client-serveur



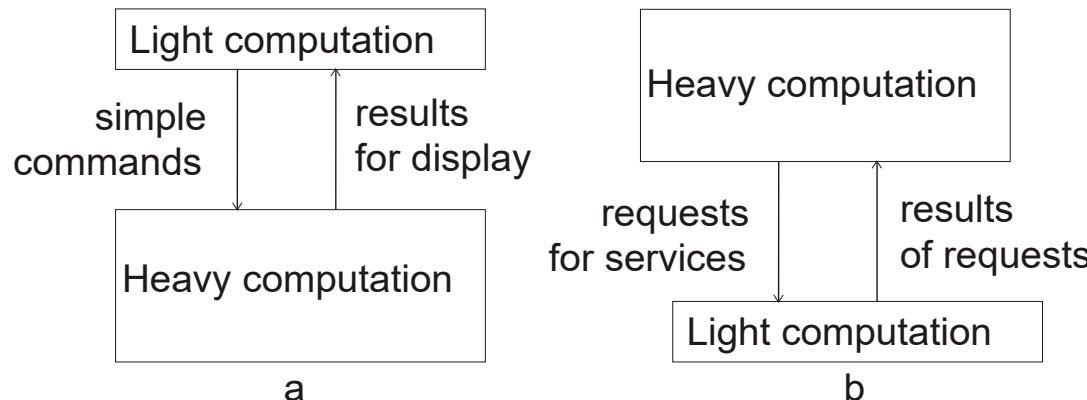
Clients légers versus clients lourds

Dans un système à *client léger (thin-client)* (a)

- Les clients sont conçus aussi petits que possible
- Le gros du travail est effectué par le serveur
- Les clients peuvent être téléchargés rapidement

Dans un système à *client lourd (fat-client)* (b)

- Autant de travail que possible est effectué localement par le client
- Le serveur peut ainsi gérer plus de connexions simultanées



Protocoles de communications

- Les messages envoyés par le client forment un *langage*.
 - Le serveur doit être conçu de façon à comprendre ce langage
- Les messages envoyés par le serveur forment aussi un *langage*
 - Le client doit être conçu de façon à comprendre ce langage
- Lorsque le client et le serveur communiquent ensemble, ils ont une conversation en utilisant ces 2 langages
- Ces deux langages et les règles de conversation mis ensemble s'appellent le *protocole*

Tâches à accomplir dans le développement d'applications client-serveur

1. Déterminer les **tâches** que doit réaliser le système client-serveur
2. Déterminer **comment le travail devra être distribué**
3. Concevoir les **messages** à être envoyés
4. Concevoir les mécanismes
 1. d'initialisation
 2. de gestion les connexions
 3. d'envoie et de réception de messages
 4. de terminaison

Avantages des systèmes client-serveur

- Le travail peut être distribué à travers plusieurs machines
- Les clients peuvent accéder aux services d'un serveur à distance
- Le programme client et le programme serveur peuvent être conçus séparément
- La structure de chacun s'en trouve souvent simplifié
- Toutes les données peuvent être centralisées au serveur
- A l'opposé, les données peuvent être géographiquement distribuées à travers les différents serveurs
- Le serveur peut être accédé simultanément par plusieurs clients
- Différents clients compétiteurs peuvent accéder au même serveur

3.5 Technologies requises afin de concevoir un système client-serveur

Protocole Internet (IP)

- Achemine les messages d'une machine à l'autre
- Les messages plus long sont habituellement subdivisés en petits morceaux

Protocole de contrôle de la transmission (TCP)

- Gère les *connexions* entre deux ordinateurs
- De cette façon plusieurs messages IP peuvent être échangés
- Assure la bonne réception des messages

Un hôte possède une adresse *IP* et un *nom*

- Plusieurs serveur peuvent exécuter sur le même hôte
- Chaque serveur est identifié par un numéro de port (0 to 65535).
- Afin d'établir une connexion avec un serveur, un client doit connaître le nom de l'hôte et le numéro de port

Établir une connexion en Java

Le paquetage `java.net`

- Permet la création d'une connexion TCP/IP entre deux applications

Avant qu'une connexion soit établie, le serveur doit se mettre à l'écoute sur l'un des ports disponibles:

```
ServerSocket serverSocket = new  
    ServerSocket(port);  
Socket clientSocket = serverSocket.accept();
```

Le client peut alors demander à se connecter au serveur:

```
Socket clientSocket= new Socket(host, port);
```

Échanger de l'information en Java

- Chaque programme utilise une instance des classes:
 - `InputStream` afin de recevoir des messages
 - `OutputStream` afin d'envoyer des messages
 - Celles-ci se trouvent dans le paquetage `java.io`

```
output = clientSocket.getOutputStream();
```

```
input = clientSocket.getInputStream();
```

Envoyer et recevoir des messages

- **sans aucun filtres (par octets)**

```
output.write(msg);  
msg = input.read();
```

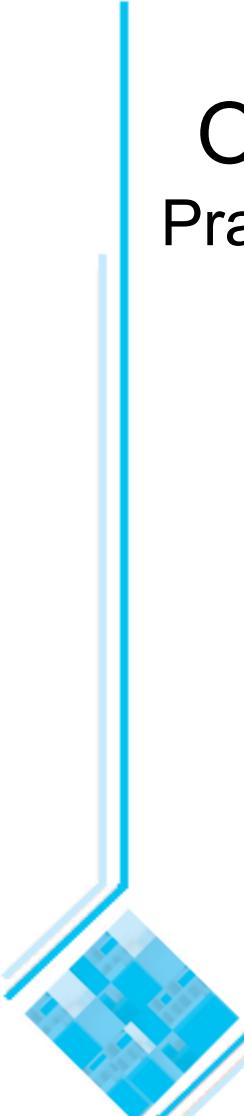
- **en utilisant les filtres DataInputStream / DataOutputStream**

```
output.writeDouble(msg);  
msg = input.readDouble();
```

- **en utilisant les filtres**

ObjectInputStream / ObjectOutputStream

```
output.writeObject(msg);  
msg = input.readObject();
```



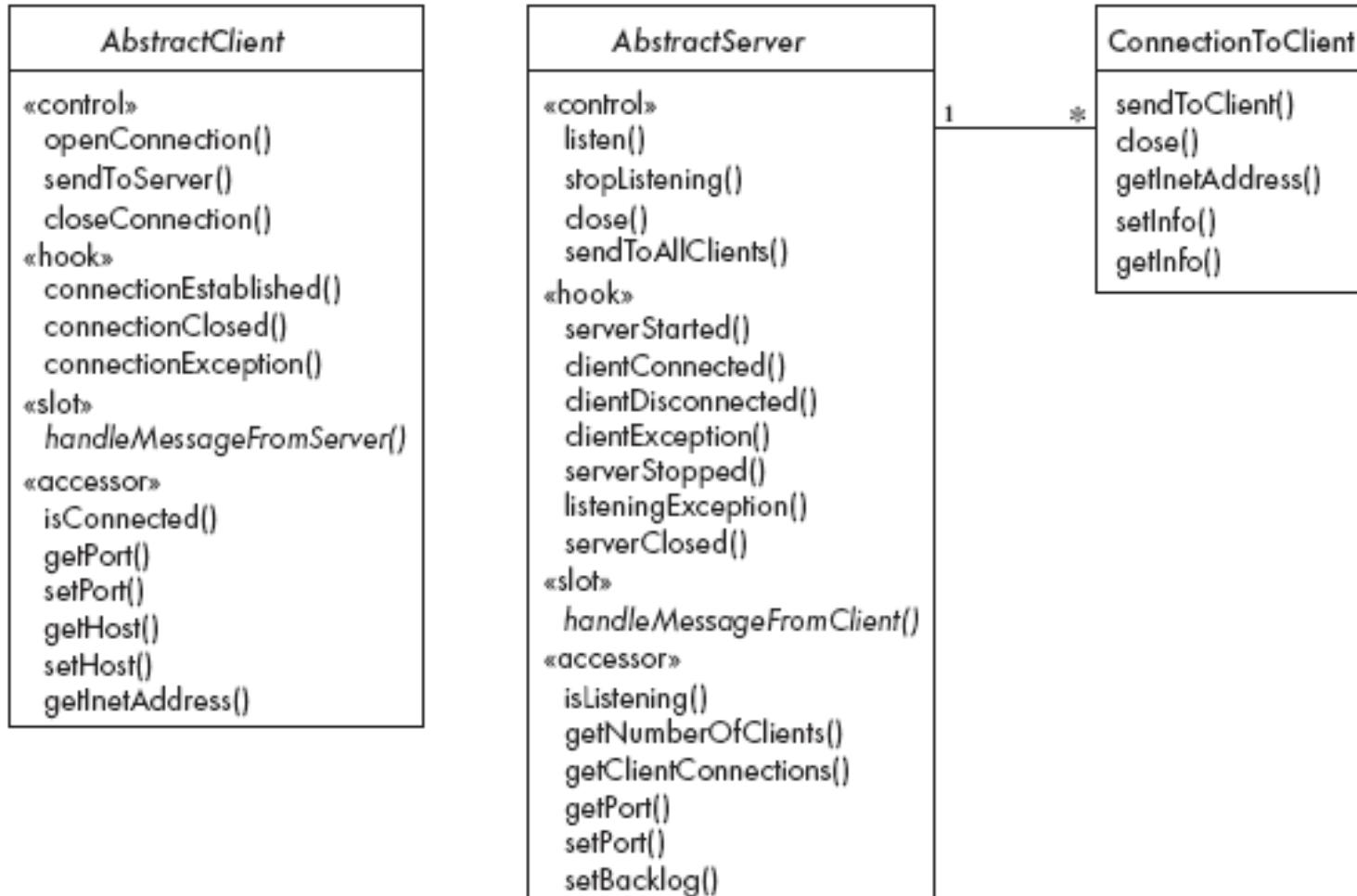
Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 3B:

Object Client-Server Framework

3.6 Le cadre d'application OCSF (Object Client-Server Framework)



Utilisation de OCSF

Les classes de OCSF, comme de tout autre cadres d'applications, ne doivent pas être modifiées

Il faut plutôt:

- Créer des sous-classes des classes abstraites présentes dans OCSF
- Appeler les méthodes disponibles dans OCSF
- Redéfinir certaines méthodes prévues à cette fin

3.7 Le côté Client

Est composé d'une seule classe: AbstractClient

- Une sous-classe de celle-ci doit être créée
 - Cette sous-classe doit proposer une définition pour la méthode handleMessageFromServer
 - Il s'agit de décrire les actions qui doivent être prises lorsqu'un message est reçu en provenance du serveur
- Implemente l'interface Runnable
 - Ce qui implique qu'elle inclut une méthode run
 - Celle-ci contient la boucle de gestion des messages

L'interface publique de AbstractClient

Méthodes de contrôle:

- **openConnection**
- **closeConnection**
- **sendToServer**

Méthodes d'accès:

- **isConnected**
- **getHost**
- **setHost**
- **getPort**
- **setPort**
- **getInetAddress**

Les méthodes de rappel de AbstractClient

Cette classe inclut des méthodes de rappel (*callback*) qui sont appelées automatiquement par le cadre

Certaines de ces méthodes peuvent être redéfinies au besoin:

- **connectionEstablished**
- **connectionClosed**

Une autre doit absolument être définie:

- **handleMessageFromServer**

L'utilisation de `AbstractClient`

- Créer une sous-classe de `AbstractClient`
- Définir la méthode ouverte
`handleMessageFromServer`
- Écrire les instructions permettant de:
 - Créer une instance de la sous-classe conçue
 - Appeler `openConnection`
 - Envoyer des messages au server en utilisant la méthode `sendToServer`
- Définir, si il y a lieu, la méthode `connectionClosed`
- Définir, si il y a lieu, la méthode `connectionException`

L'intérieur de AbstractClient

Les variables d'instance:

- Une variable de type **Socket** contenant l'information à propos de la connexion avec le serveur
- Deux flots de communication:
ObjectOutputStream et **ObjectInputStream**
- Un fil d'exécution (**thread**) qui exécute d'à partir de la méthode run de la classe **AbstractClient**
- Deux variables contenant le nom et le numéro de l'hôte

3.8 Le côté Serveur

Constitué de deux classes:

- Une pour gérer le fil d'exécution étant à l'écoute de nouvelle demande de connexion (**AbstractServer**)
- Une pour l'ensemble des fils d'exécution gérant les échanges avec les clients connectés (**ConnectionToClient**)

L'interface publique de AbstractServer

Méthodes de contrôle:

- **listen**
- **stopListening**
- **close**
- **sendToAllClients**

Méthodes d'accès:

- **isListening**
- **getClientConnections**
- **getPort**
- **setPort**
- **setBacklog**

Les méthodes de rappel de AbstractServer

Certaines de ces méthodes peuvent être redéfinies au besoin:

- **serverStarted**
- **clientConnected**
- **clientDisconnected**
- **clientException**
- **serverStopped**
- **listeningException**
- **serverClosed**

Une autre doit absolument être définies:

- **handleMessageFromClient**

L'interface publique de ConnectionToClient

Méthodes de contrôle:

- **sendToClient**
- **close**

Méthodes d'accès:

- **getInetAddress**
- **setInfo**
- **getInfo**

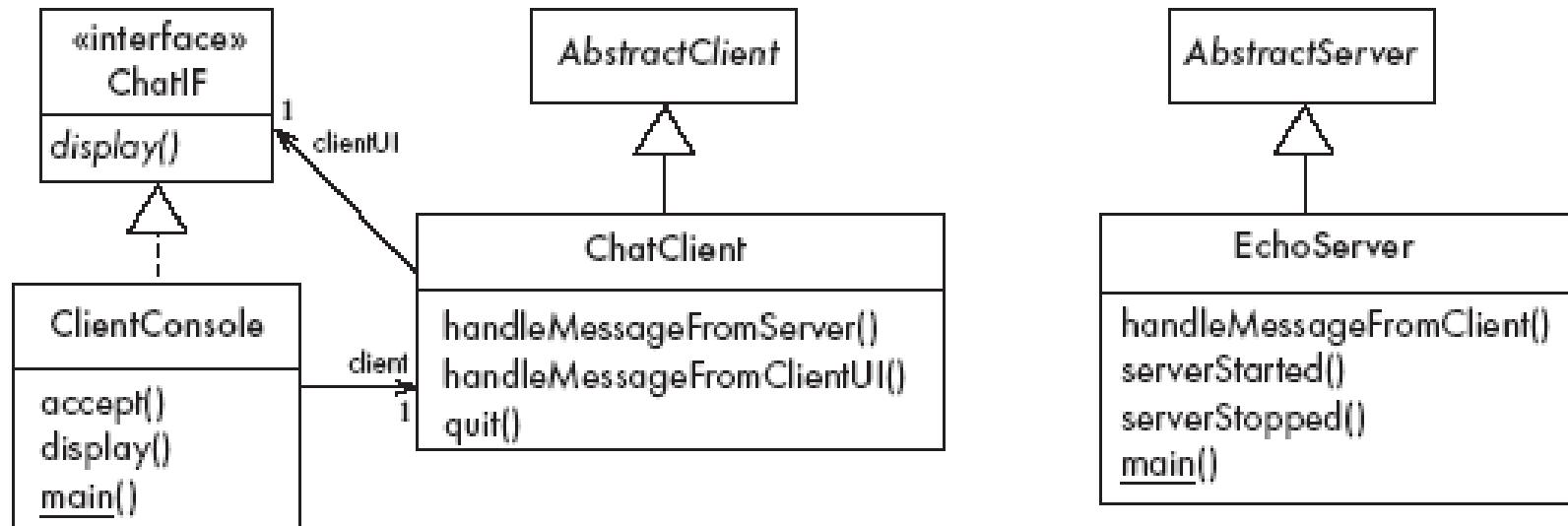
L'utilisation de `AbstractServer` et de `ConnectionToClient`

- Créer une sous-classe de `AbstractServer`
- Définir la méthode ouverte
`handleMessageFromClient`
- Écrire les instructions permettant de:
 - Créer une instance de la sous-classe conçue
 - Appeler la méthode `listen`
 - Envoyer des messages aux clients en utilisant:
 - `getClientConnections` et `sendToClient`
 - ou encore `sendToAllClients`
- Définir une ou l'autre des méthodes de rappel offertes

L'intérieur de AbstractServer et ConnectionToClient

- Les méthodes **setInfo** et **getInfo** utilise la classe Java **HashMap**
- Plusieurs des méthodes côté serveur ont dû être synchronisées (mot-clé **synchronized**)
- L'ensemble des instances de **ConnectionToClient** sont contenues dans une classe spéciale appelée **ThreadGroup**
- Le serveur doit faire une pause à toutes les 500ms afin de vérifier si la méthode **stopListening** a été appelée
 - Si non, il se remet en mode d'attente

3.9 Une application simple de clavardage



ClientConsole pourrait être remplacé par ClientGUI

Le serveur

EchoServer est une sous-classe de **AbstractServer**

- La méthode **main** en crée une instance et en lance l'exécution
 - Elle gère les connexions avec les clients jusqu'à ce que le serveur soit stoppé
- Les trois méthodes de rappel ne font qu'afficher un message à l'utilisateur
 - **handleMessageFromClient**, **serverStarted** et **serverStopped**
- Le méthode **handleMessageFromClient** appelle le service **sendToAllClients**
 - Ce qui renvoie le message reçu à tous les clients

Le cœur de EchoServer

```
public void handleMessageFromClient  
    (Object msg, ConnectionToClient client)  
{  
    System.out.println(  
        "Message received: "  
        + msg + " from " + client);  
    this.sendToAllClients (msg);  
}
```

Le client

Lorsque le programme client démarre, il crée des instances de deux classes:

- **ChatClient**
 - Une sous-classe de **AbstractClient**
 - Redéfinit la méthode **handleMessageFromServer**
 - Qui à son tour appelle la méthode **display** de l'interface usager
- **ClientConsole**
 - L'interface usager réalisant l'interface **ChatIF**
 - Définit la méthode **display** qui affiche les messages sur la console
 - Accepte les entrées de l'utilisateur via la méthode **accept**
 - Envoie toutes les entrées à **ChatClient** via la méthode **handleMessageFromClientUI**
 - Celle-ci en retour appelle la méthode **sendToServer**

Le cœur de ChatClient

```
public void handleMessageFromClientUI( String message)
{
    try
    {
        sendToServer(message);
    }
    catch(IOException e)
    {
        clientUI.display (
            "Could not send message. " +
            "Terminating client.");
        quit();
    }
}
```

Le cœur de ChatClient - suite

```
public void handleMessageFromServer(Object msg)
{
    clientUI.display(msg.toString());
}
```

3.12 Difficultés et risques lié à la réutilisation

- **La réutilisation de composantes de pauvre qualité peut être très coûteuse**
 - S'assurer que le développement de ces composantes se fait conformément aux bonnes pratiques du génie logiciel*
 - Qu'un support est offert aux utilisateurs de ces composantes*
- **La compatibilité peut elle être assurée?**
 - Éviter l'emploi d'éléments peu communs ou inusités*
 - Réutiliser des technologies qui sont aussi réutilisées par d'autres, i.e. prévues à cette fin*

Difficultés et risques lié au développement de composantes réutilisables

- Il s'agit d'un investissement incertain
 - Planifier le développement de la même façon que pour un produit destiné à un client*
- Faire face à la réticence usuel des programmeurs à utiliser des composantes qu'ils n'ont pas conçus
 - Donner confiance en:*
 - *Garantissant un bon support*
 - *Assurant la qualité du produit*
 - *Répondant aux besoins des réutilisateurs*

Difficultés et risques lié au développement de composantes réutilisables

- **Compétition**
 - L'utilité et la qualité d'une composante réutilisable est son principal atout*
- **Divergence**
 - Concevoir d'une façon aussi générique que possible*



Difficultés et risques lié au développement d'application client-serveur

- Sécurité
 - Les aspects de sécurité constituent toujours un problème critique*
 - Considérer l'usage de l'encryptage, de murs coupe-feu*
- Besoin de maintenance adaptative
 - S'assurer que les client et serveurs demeurent compatibles à mesure que les version évolues*

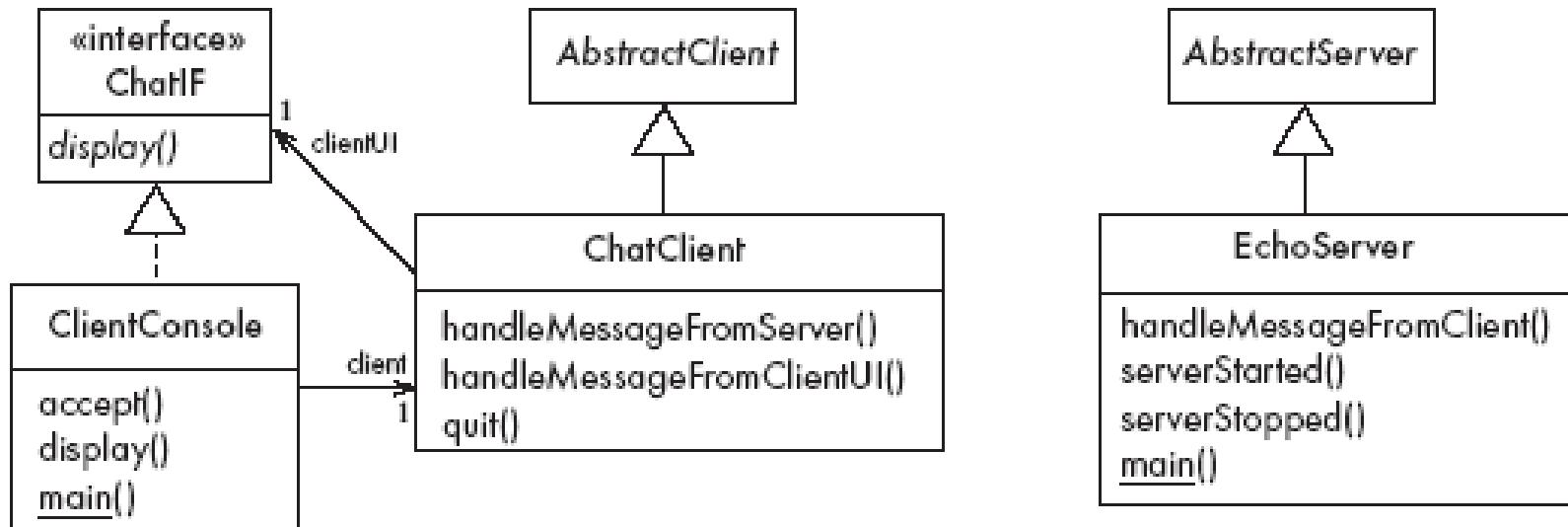
Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 3C:

Chat Application

3.9 Une application simple de clavardage



ClientConsole pourrait être remplacé par ClientGUI

Le serveur

EchoServer est une sous-classe de **AbstractServer**

- La méthode **main** en crée une instance et en lance l'exécution
 - Elle gère les connexions avec les clients jusqu'à ce que le serveur soit stoppé
- Les trois méthodes de rappel ne font qu'afficher un message à l'utilisateur
 - **handleMessageFromClient**, **serverStarted** et **serverStopped**
- Le méthode **handleMessageFromClient** appelle le service **sendToAllClients**
 - Ce qui renvoie le message reçu à tous les clients

Le cœur de EchoServer

```
public void handleMessageFromClient  
    (Object msg, ConnectionToClient client)  
{  
    System.out.println(  
        "Message received: "  
        + msg + " from " + client);  
    this.sendToAllClients (msg);  
}
```

Le client

Lorsque le programme client démarre, il crée des instances de deux classes:

- **ChatClient**
 - Une sous-classe de **AbstractClient**
 - Redéfinit la méthode **handleMessageFromServer**
 - Qui à son tour appelle la méthode **display** de l'interface usager
- **ClientConsole**
 - L'interface usager réalisant l'interface **ChatIF**
 - Définit la méthode **display** qui affiche les messages sur la console
 - Accepte les entrées de l'utilisateur via la méthode **accept**
 - Envoie toutes les entrées à **ChatClient** via la méthode **handleMessageFromClientUI**
 - Celle-ci en retour appelle la méthode **sendToServer**

Le cœur de ChatClient

```
public void handleMessageFromClientUI( String message)
{
    try
    {
        sendToServer(message);
    }
    catch(IOException e)
    {
        clientUI.display (
            "Could not send message. " +
            "Terminating client.");
        quit();
    }
}
```

Le cœur de ChatClient - suite

```
public void handleMessageFromServer(Object msg)
{
    clientUI.display(msg.toString());
}
```

3.12 Difficultés et risques lié à la réutilisation

- **La réutilisation de composantes de pauvre qualité peut être très coûteuse**
 - Il faut s'assurer que le développement de ces composantes se fait conformément aux bonnes pratiques du génie logiciel*
 - Il faut s'assurer qu'un support est offert aux utilisateurs de ces composantes*
- **La compatibilité peut elle être assurée?**
 - Éviter l'emploi d'éléments peu communs ou inusités*
 - Réutiliser des technologies qui sont aussi réutilisées par d'autres, i.e. prévues à cette fin*

Difficultés et risques lié au développement de composantes réutilisables

- Il s'agit d'un investissement incertain
 - Planifier le développement de la même façon que pour un produit destiné à un client*
- Faire face à la réticence usuel des programmeurs à utiliser des composantes qu'ils n'ont pas conçus
 - Donner confiance en:*
 - *Garantissant un bon support*
 - *Assurant la qualité du produit*
 - *Répondant aux besoins des réutilisateurs*

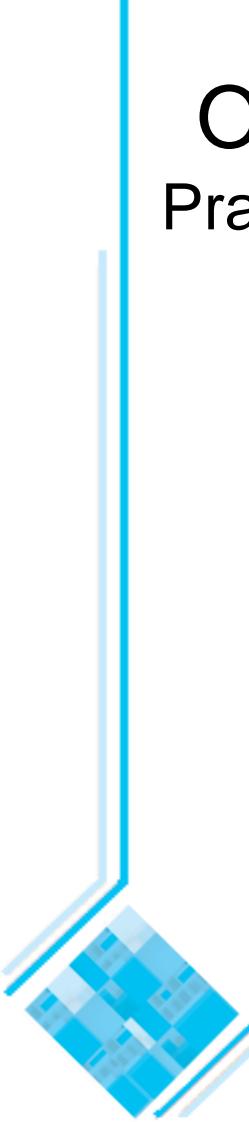
Difficultés et risques lié au développement de composantes réutilisables

- **Compétition**
 - L'utilité et la qualité d'une composante réutilisable est son principal atout*
- **Divergence**
 - Concevoir d'une façon aussi générique que possible*



Difficultés et risques lié au développement d'application client-serveur

- Sécurité
 - Les aspects de sécurité constituent toujours un problème critique*
 - Considérer l'usage de l'encryptage, de murs coupe-feu*
- Besoin de maintenance adaptative
 - S'assurer que les client et serveurs demeurent compatibles à mesure que les version évolues*



Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 4:

Developing Requirements

4.1 Analyse de domaine

Il s'agit du processus par lequel un ingénieur logiciel peut faire l'apprentissage du domaine de l'application:

- Le *domaine* correspond au champs d'application dans lequel le logiciel sera utilisé.
- Un expert de ce domaine est une personne ayant une connaissance approfondie de ce domaine
- Cette étude a pour but ultime de mieux comprendre le problème à résoudre

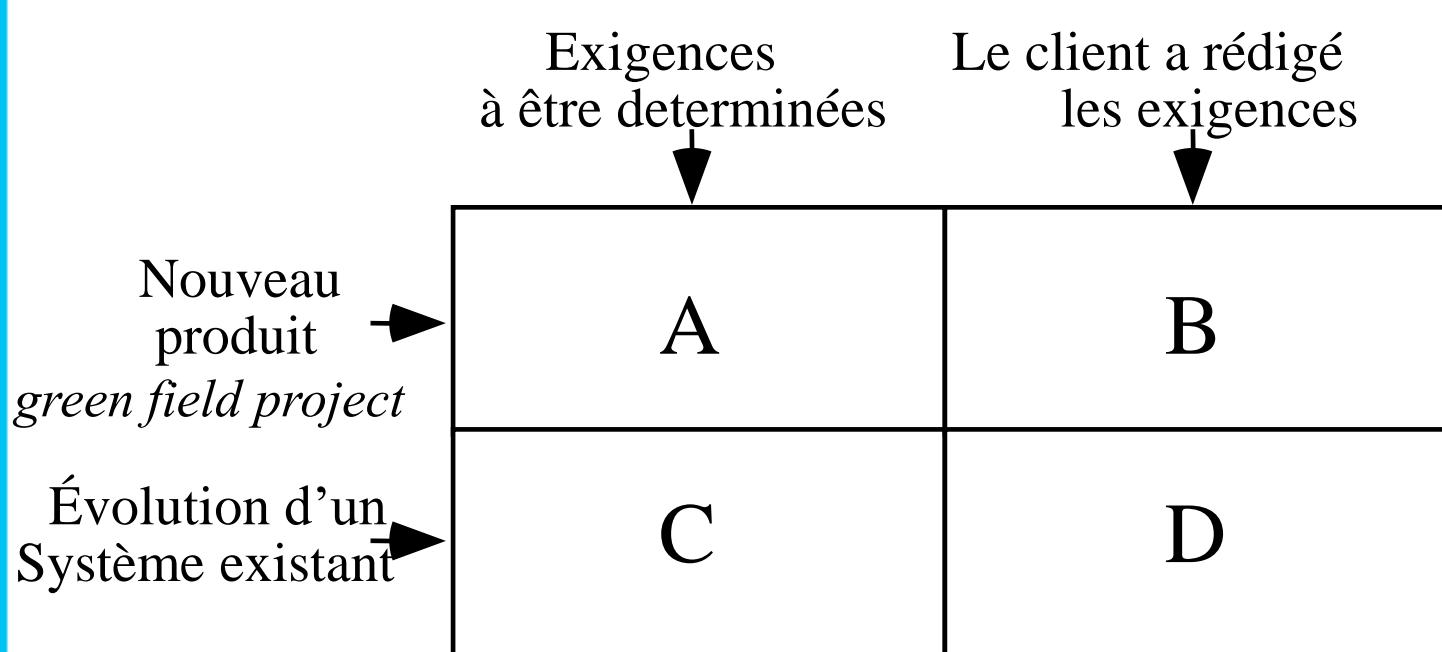
Bénéfices découlant de l'analyse de domaine:

- Accélère le développement
- Permet le développement d'un meilleur système
- Permet d'anticiper les possibles extensions

Document d'analyse de domaine

- A. Introduction**
- B. Glossaire**
- C. Connaissances générales**
- D. Les clients et utilisateurs**
- E. L'environnement**
- F. Tâches et procédures en usage**
- G. Logiciel concurrent**
- H. Similarités avec d'autres domaines**

4.2 Le point de départ



4.3 Définition du problème et de sa portée

Un problème peut être exprimé comme:

- Une difficulté auquel se voit confronté le client ou les utilisateurs
- Ou une opportunité qui produira un certain bénéfice tel que une augmentation de productivité ou de ventes.

La résolution du problème mène au développement d'un logiciel

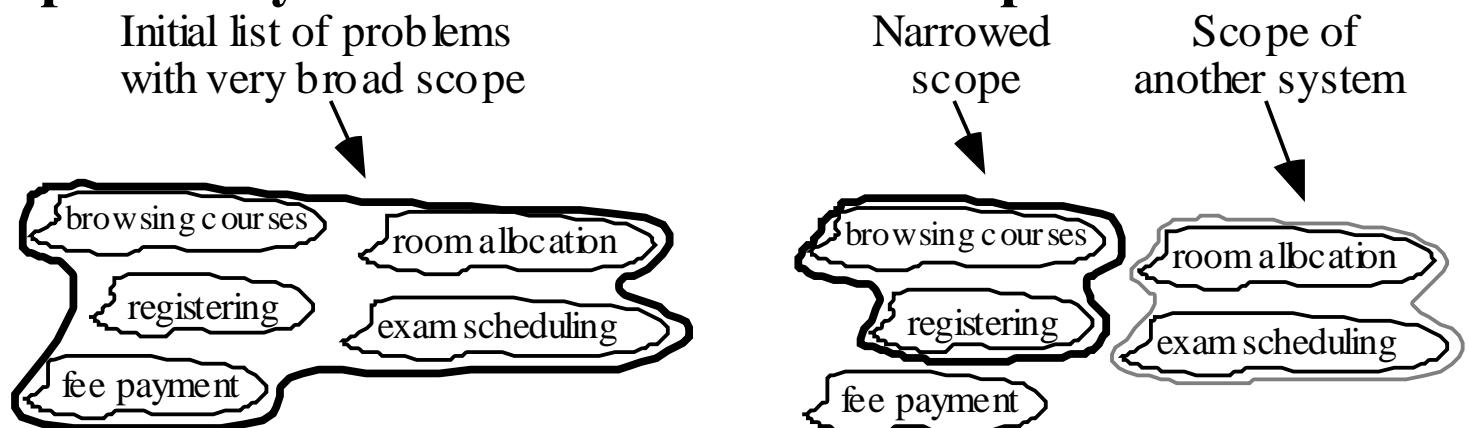
Un bon énoncé de problème doit être clair et concis

Définir la portée

Cerner la portée en définissant de façon plus précise le problème à résoudre

- Lister toutes les choses que le système devrait avoir à faire
 - Exclure le plus de choses possibles pour réduire la complexité du problème
 - Établir des buts plus généraux si le problème devient trop simple

Exemple: un système automatisé d'inscription universitaire



4.4 Qu'est-ce qu'une exigence?

Un énoncé décrivant soit:

- 1) Ce que le système doit faire
 - 2) Une contrainte concernant le développement du système
-
- Cet énoncé doit contribuer à résoudre le problème du client
 - Et doit avoir été accepté par toutes les parties prenantes

Un ensemble d'exigences est un *cahier des charges* (*requirements document*).

4.5 Types d'exigences

Exigences fonctionnelles

- Décrit ce que le système doit faire

Exigences de qualité

- *Contraintes* sur le niveau de qualité que le design doit rencontrer

Exigences de plate-forme

- *Contraintes* sur l'environnement et la technologie

Exigences de processus

- *Contraintes* sur la gestion du projet et la méthodologie de développement

Exigences fonctionnelles

- Quelles doivent être les *entrées* du système
- Quelles *sorties* le système doit-il produire
- Quelle sont les *données* qui devront être stockées pour usage par d'autres systèmes
- Quels sont les *calculs* à effectuer
- La *mise en marche* et la *synchronisation* de tous ces éléments

Exigences de qualité

Doivent être vérifiables

- Temps de réponse
- Rendement, débit
- Utilisation des ressources
- Fiabilité
- Disponibilité
- Restauration après pannes
- Facilité de maintenance et d'amélioration
- Facilité de réutilisation

4.6 Les cas d'utilisation

Un *cas d'utilisation (use case)* est une séquence typique d'actions qu'entreprend un utilisateur afin d'accomplir une tâche donnée

- L'objectif d'une *analyse de cas* est de modéliser le système
 - ... du point de vue de la façon par laquelle l'utilisateur interagit avec le système
 - ... afin d'accomplir ses objectifs
- Il s'agit-là de l'une des activités les plus importantes pour la cueillette et l'analyse des exigences du système
- Un *modèle des cas d'utilisations* consiste en
 - un ensemble de cas d'utilisation
 - une description ou un diagramme expliquant comment ils sont inter-reliés

Les cas d'utilisations

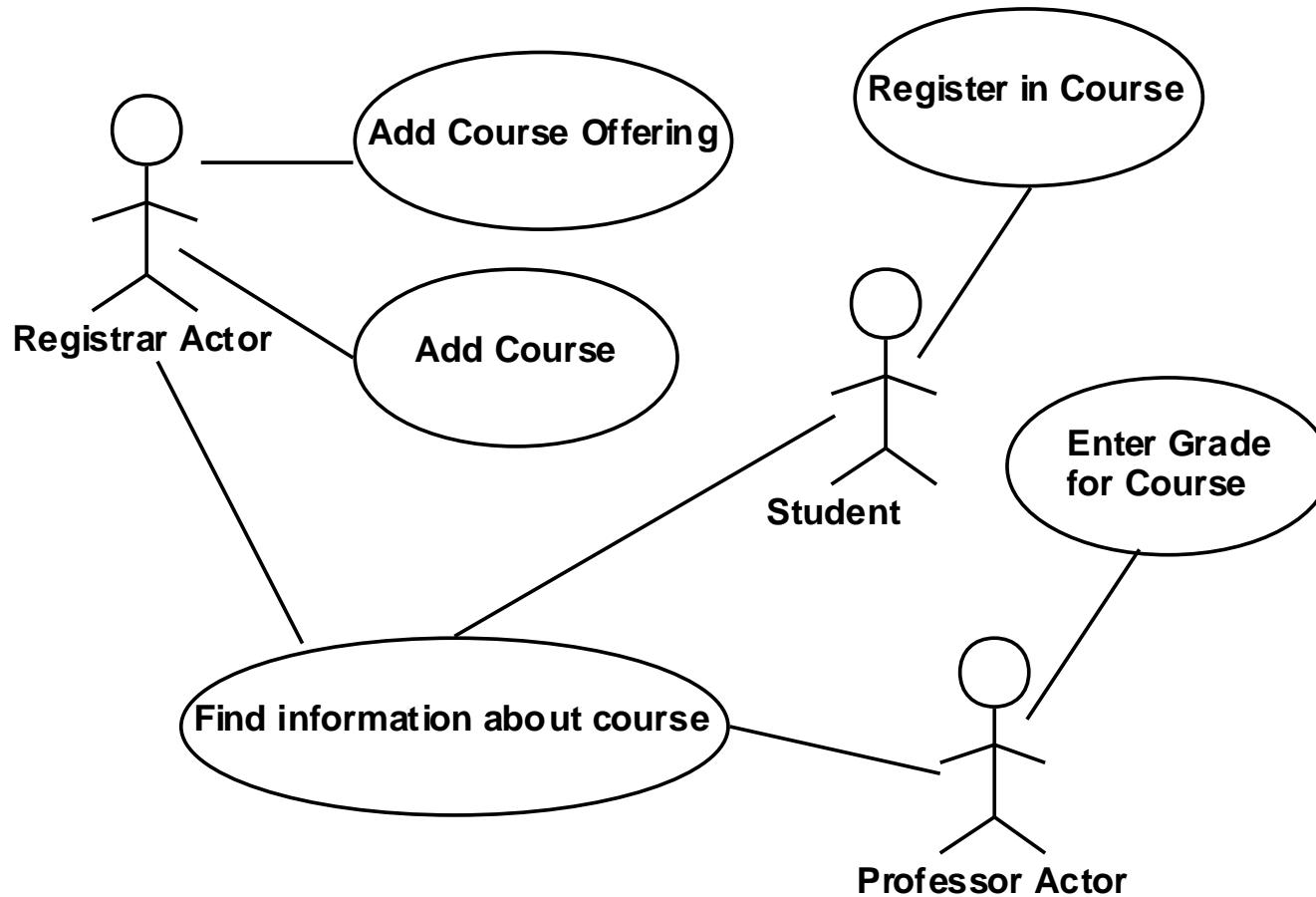
- En général, un cas d'utilisation doit couvrir *l'ensemble des étapes à suivre* dans l'accomplissement d'une tâche donnée
- Le cas d'utilisation doit décrire *l'interaction* de l'utilisateur avec le système
 - et non les opérations que doit réaliser le système.
- Le cas d'utilisation devrait être écrit, dans la mesure du possible, d'une façon *indépendante* de toute interface utilisateur
- Un cas d'utilisation ne doit inclure que les interactions avec le système

Synonymes: Cas-type, cas-type d'utilisation, *use case...*

Comment décrire un cas d'utilisation?

- A. Nom:** Une appellation courte et descriptive
- B. Acteurs:** Énumérer les acteurs impliqués
- C. Buts:** Expliquer ce que les acteurs veulent accomplir
- D. Préconditions:** Décrire l'état du système avant l'exécution de ce cas d'utilisation
- E. Description:** Une courte description informelle
- F. Référence:** Autre cas d'utilisation apparentés
- G. Déroulement:** Décrire chacune des étapes, sur 2 colonnes
- H. Postconditions:** Décrire l'état du système après l'exécution de ce cas d'utilisation

Diagrammes de cas d'utilisation



Extensions

- Utilisés afin de spécifier des interactions *optionnelles* tenant compte des cas *exceptionnels*
- En créant ainsi un cas d'utilisation d'extension, la description principale peut demeurer simple

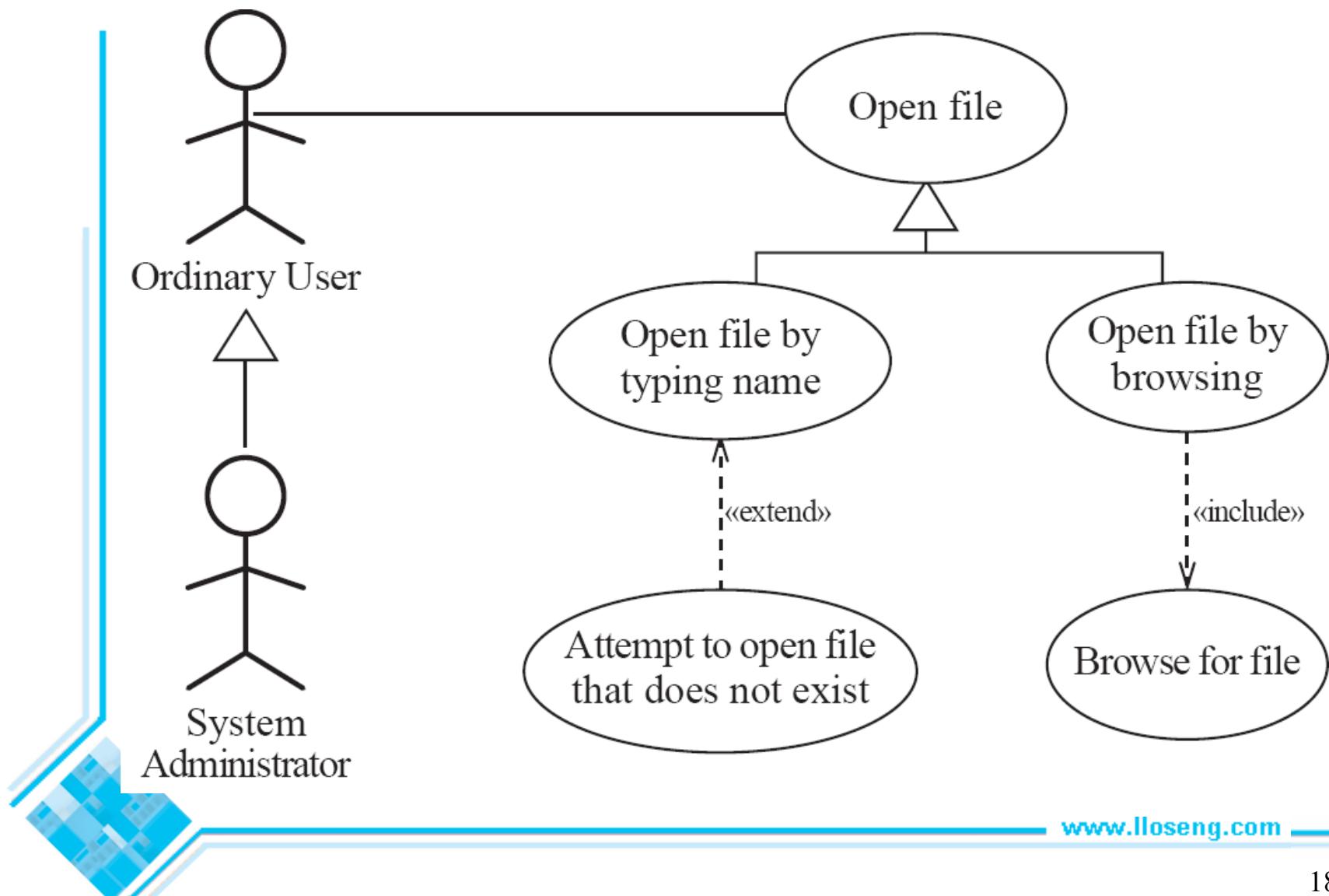
Généralisations

- Similaire aux super-classes dans les diagrammes de classes
- Un cas d'utilisation généralisé représente plusieurs cas d'utilisations similaires
- La spécialisation d'un cas d'utilisation fournit les détails spécifiques relatifs à un de ces cas d'utilisations similaires

Inclusions

- Sert à décrire les points communs contenus dans plusieurs cas d'utilisations différents
- Sont donc inclus dans d'autres cas d'utilisations
 - Même des cas d'utilisations très différents peuvent partager un certain nombre d'actions identiques
 - Permet d'éviter la répétition de ces détails dans chacun des cas d'utilisations
- Représente l'exécution d'une tâche de plus bas niveau pour accomplir un but aussi de plus bas niveau

Exemple de généralisation, d'extension et d'inclusion



Exemple de description d'un cas d'utilisation

Use case: Open file

Related use cases:

Generalization of:

- Open file by typing name
- Open file by browsing

Steps:

Actor actions

1. Choose 'Open...' command
3. Specify filename
4. Confirm selection

System responses

2. File open dialog appears
5. Dialog disappears

Exemple (suite)

Use case: Open file by typing name

Related use cases:

Specialization of: Open file

Steps:

Actor actions

1. Choose ‘Open...’ command
- 3a. Select text field
- 3b. Type file name
4. Click ‘Open’

System responses

2. File open dialog appears
5. Dialog disappears

Exemple (suite)

Use case: Open file by browsing

Related use cases:

Specialization of: Open file

Includes: Browse for file

Steps:

Actor actions

1. Choose ‘Open...’ command
3. Browse for file
4. Confirm selection

System responses

2. File open dialog appears
5. Dialog disappears

Exemple (suite)

Use case: Browse for file (inclusion)

Steps:

Actor actions

1. If the desired file is not displayed, select a directory
3. Repeat step 1 until the desired file is displayed
4. Select a file

System responses

2. Contents of directory is displayed

Exemple (suite)

Use case: Attempt to open file that does not exist

Related use cases:

Extension of: Open file by typing name

Actor actions

1. Choose ‘Open...’ command
- 3a. Select text field
- 3b. Type file name
4. Click ‘Open’

6. Correct the file name
7. Click ‘Open’

System responses

2. File open dialog appears

5. System indicates that file does not exist

- 8 Dialog disappears

Le processus de modélisation: choisir le cas d'utilisation central

- Souvent l'un des cas d'utilisations (ou quelques uns) peut être identifié comme central au système
 - Le système peut alors être conçu autour de ce cas d'utilisation particulier
- Il existe d'autres raisons de se concentrer sur un cas d'utilisation en particulier:
 - Certains cas d'utilisations présentent un risque élevé dont la réalisation peut-être problématique
 - Certains cas d'utilisations présentent une valeur commerciale ou politique importante

Les bénéfices liés à un développement fondé sur les cas d'utilisations

- Cela aide à mieux définir la portée du système
- Cela aide à mieux planifier le développement du logiciel
- Cela permet de définir et de valider les exigences
- Cela permet de définir les scénarios de tests
- Cela permet de mieux structurer les manuels d'utilisation



Histoires d'utilisateurs vs cas d'utilisation

Le terme « histoires d'utilisateur » est utilisé dans le développement agile

- Conceptuellement très similaire aux cas d'utilisation
- Généralement moins formel

4.7 Quelques techniques pour la cueillette et l'analyse des exigences

Observation

- Lire tous les documents disponibles et discuter avec les utilisateurs
- Observer des utilisateurs potentiel à leur travail

Entrevues

Séance de remue-méninge (avec les parties prenantes)

Prototypage

- Le plus commun: une maquette de l'interface usager

Analyse de cas d'utilisation

4.9 Révision des exigences

Chacune des exigences

- Doit avoir un *bénéfice* qui l'emporte sur les coûts qu'elle engendre
- Doit être *importante* dans la solution du problème
- Doit être exprimée d'une façon *claire, concise et consistante*
- Doit être *non-ambiguë*
- Doit être en *concordance* avec les standards et pratiques
- Doit mener à un système de *qualité*
- Doit être *réaliste* considérant les ressources disponibles
- Doit être *vérifiable*
- Ne doit *pas sur-constrainer* le système

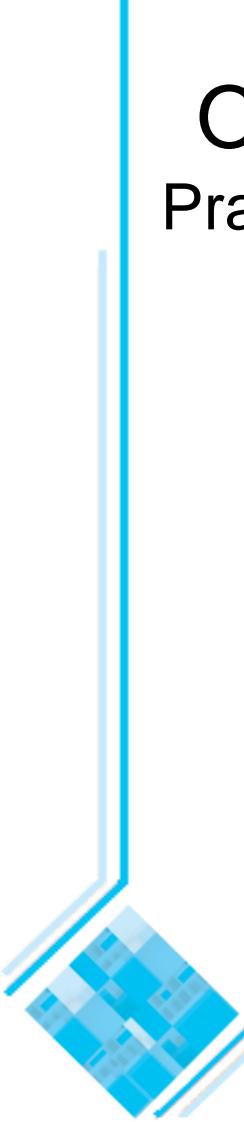
4.10 Faire face au changements

Les exigences changent parce que:

- Les procédures évoluent
- La technologie change
- Le problème est mieux maîtrisé

L'analyse des exigences ne s'arrête jamais

- Toujours continuer à interagir avec le client et les utilisateurs
- Tout changement doit être avantageux
 - De petits changements cosmétiques sont souvent faciles et peu coûteux
 - Des changements plus fondamentaux doivent être entrepris avec grande précaution
 - Effectuer des changements dans un système en cours de développement peut mener à une mauvaise conception et à des retard de livraison
- Certains changements sont en fait des ajouts au système
 - Éviter de rendre le système plus gros,
 - il doit simplement devenir meilleur



Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 5A:

Modelling with Classes

5.1 Qu'est-ce que UML?

Le « Unified Modelling Language » est un langage graphique standard pour la modélisation de logiciels orientés objet

- Développé par Rumbaugh, Booch et Jacobson
- Basé sur des langages précédentes qu'ils avaient chacun développé
- Ils ont travaillé ensemble à Rational Software Corporation, rachetée plus tard par IBM
 - Une grande partie du développement d'UML a été effectuée à IBM Rational Ottawa
- En 1997, l'Object Management Group (OMG) a lancé le processus de standardisation de UML

Les diagrammes UML

- Diagrammes de classes
 - décrit les **classes** et leurs **interrelations**
- Diagrammes d'interactions
 - montre le **comportement** du systèmes par l'**interaction** des objets qui le compose
- Diagramme d'états
 - montre comment le système se **comporte** de façon interne
- Diagramme de composantes et de déploiement
 - montre comment les différentes composantes du système sont **organisés physiquement et logiquement**

5.2 Éléments essentiels des diagrammes de classe de UML

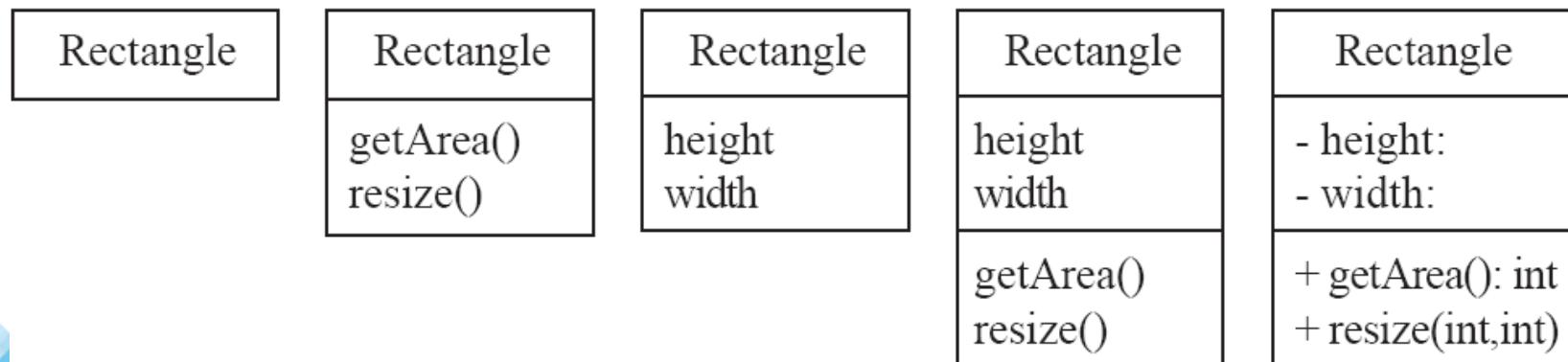
Les principaux symboles présents dans les diagrammes de classes sont:

- *Les classes*
 - Représentant les types de données disponibles
- *Les associations*
 - Représentant les liens entre les instances des classes
- *Les attributs*
 - De simples données se trouvant dans les classes et leurs instances
- *Les opérations*
 - Représentant les fonctions exécutées par les classes et leurs instances
- *Les généralisations*
 - Groupant les classes en hiérarchie d'héritage

Les classes

Une classe se représente à l'aide d'une boîte comprenant le nom de la classe

- Le diagramme peut aussi montrer les attributs et les opérations
- La signature complète d'une opération est:
operationName(parameterName: parameterType ...): returnType



5.3 Associations et Multiplicité

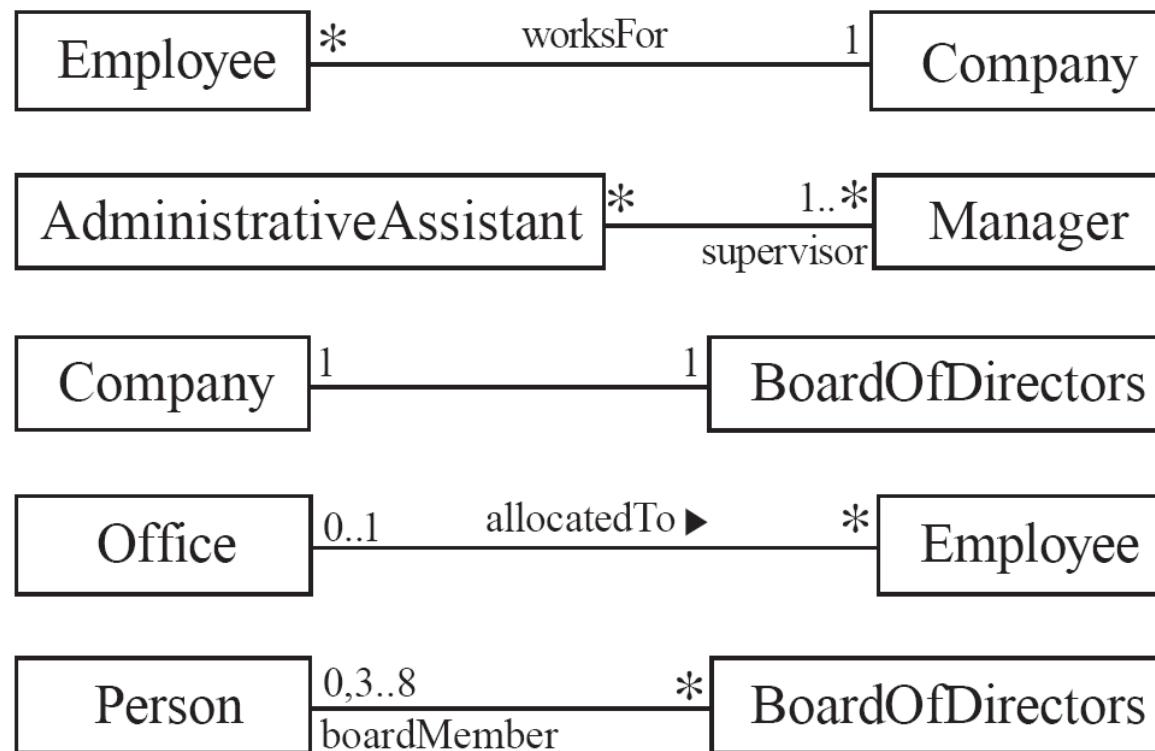
Une *association* est utilisée afin de montrer comment deux classes sont liées entre elles

- Différents symboles sont utilisés pour indiquer la *multiplicité* à chaque extrémité d'une association



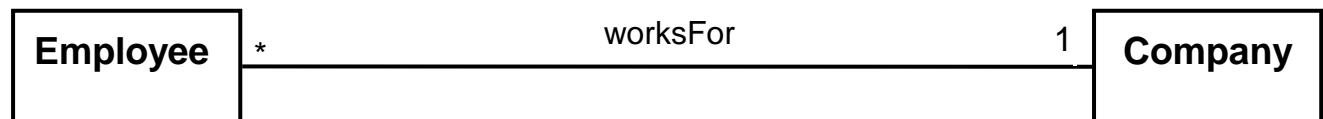
Étiqueter les associations

- Une association peut être étiquetée afin de rendre explicite la nature de cette association



Analyser et valider les associations

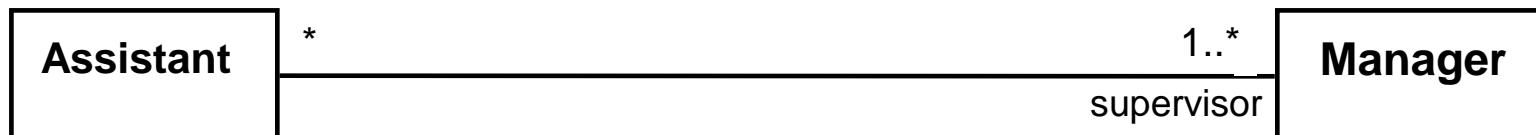
- **Une à plusieurs**
 - Une compagnie a plusieurs employés
 - Un employé ne peut travailler que pour une seule compagnie
 - Qu'en est-il des employés occupant un double emploi!
 - Une compagnie peut n'avoir aucun employé
 - Un employé associé à une compagnie travaille pour cette compagnie



Analyser et valider les associations

- **Plusieurs à plusieurs**

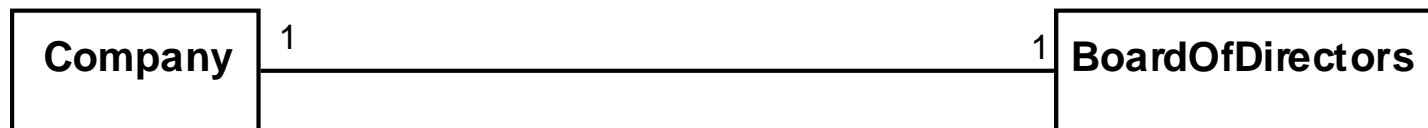
- Un(e) secrétaire peut travailler pour plusieurs superviseurs
- Un superviseur peut avoir plusieurs secrétaires
- Les secrétaires peuvent travailler en équipes
- Les superviseurs peuvent avoir recours à un groupe de secrétaires
- Certains superviseurs peuvent n'avoir aucun(e) secrétaires
- Est-il possible qu'un(e) secrétaire puisse se retrouver, ne serait-ce que temporairement, sans superviseur?



Analyser et valider les associations

- **Une à une**

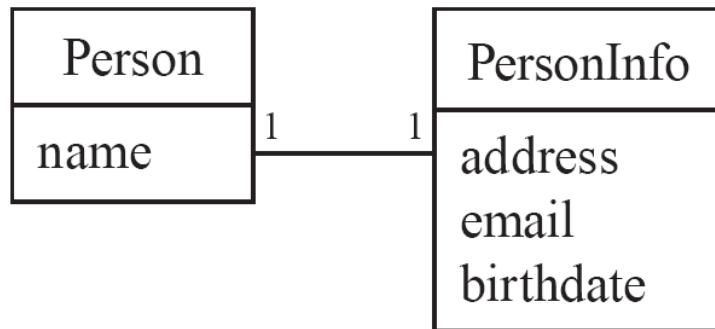
- A chaque compagnie est associé un conseil d'administration
- Un conseil d'administration gère une seule compagnie
- Une compagnie doit avoir un conseil d'administration
- Un conseil d'administration est toujours attaché à une et une seule compagnie



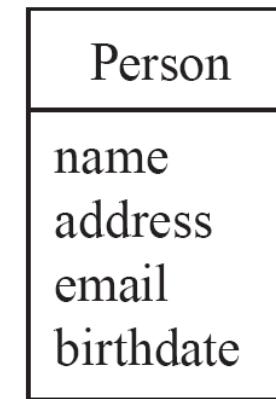
Analyser et valider les associations

Attention aux associations une-à une injustifiées

Éviter ceci

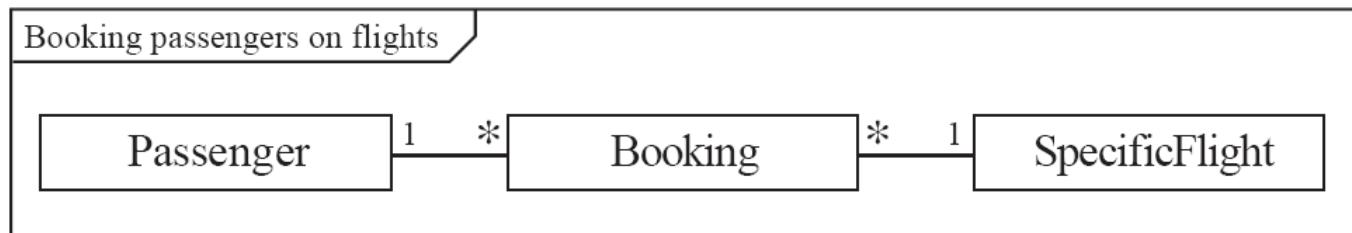


Cela est préférable



Un exemple plus complexe

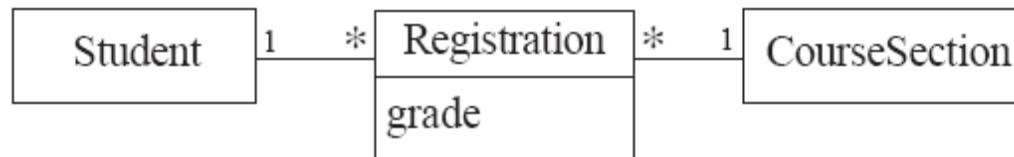
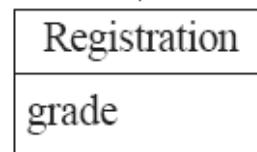
- Une réservation est pour un passager
 - Pas de réservations sans passager
 - Une réservation ne devrait jamais compter plus d'un passager
- Un passager peut effectuer plusieurs réservations
 - Un passager peut aussi n'avoir fait aucune



- Le cadre autour du diagramme est une nouvelle option de UML 2.0

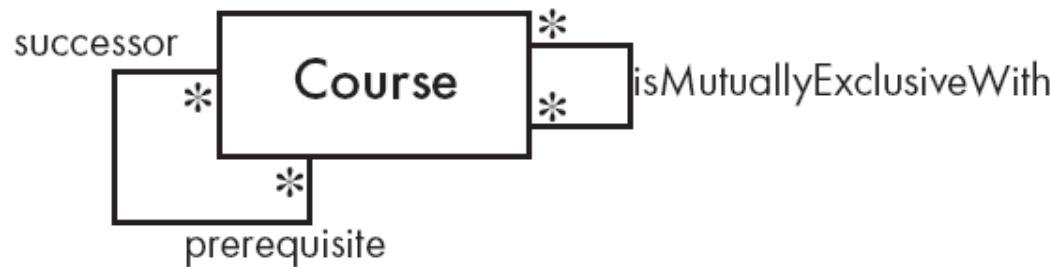
Classes d'association

- Il arrive, quelques fois, qu'un attribut ne puisse être attaché à aucune des deux classes d'une association
- Mais il peut être attaché à l'association elle-même



Association réfléchie

- Une classe peut être associée à elle-même



Association directionnelle

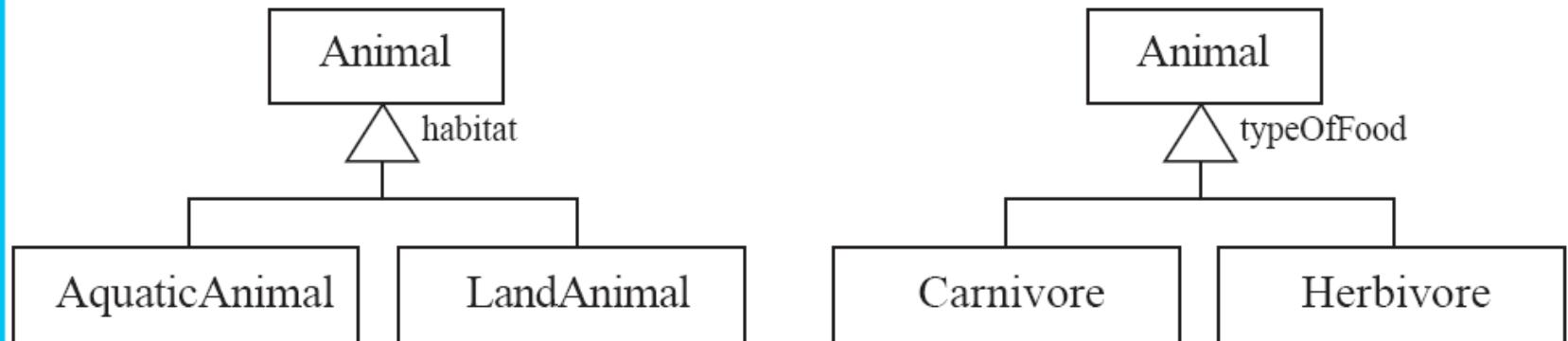
- Les associations sont, par défaut, *bi-directionnelles*
- Il est toutefois possible de donner, à l'aide d'une flèche, une direction à une association



5.4 Généralisation

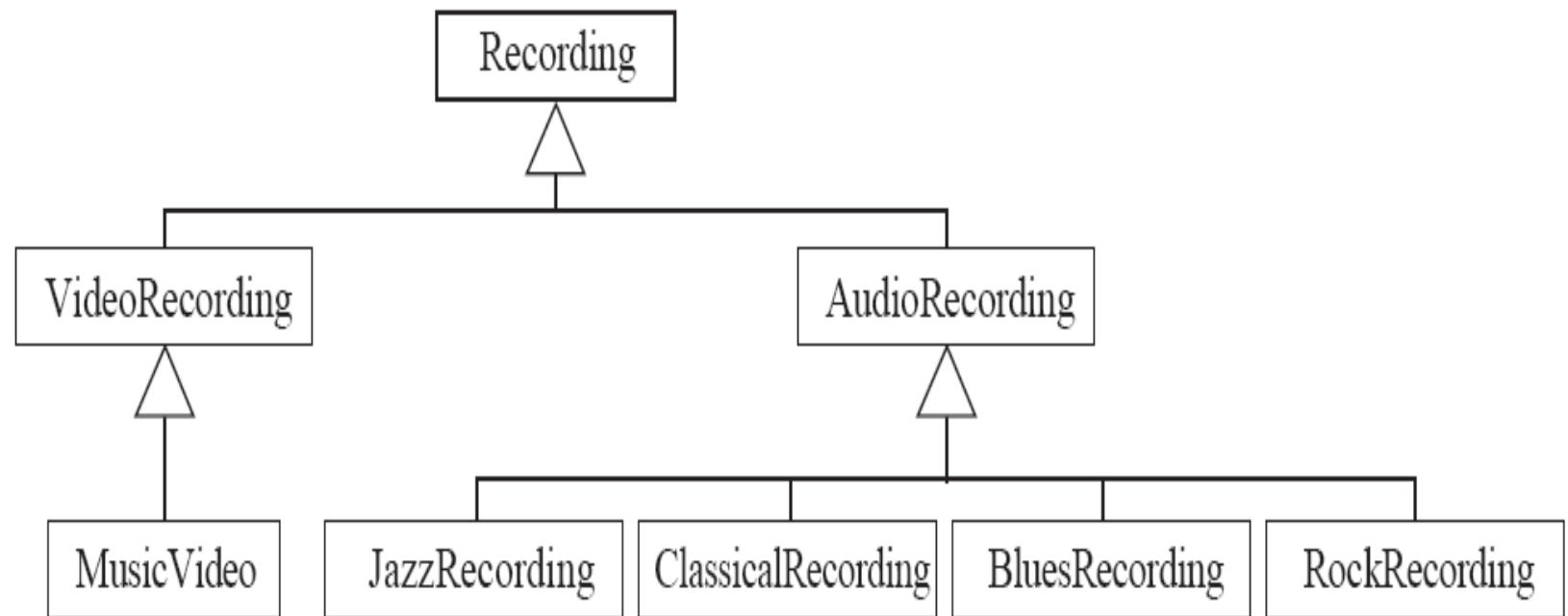
Une super-classe se spécialise en sous-classes

- Le *discriminant* est une étiquette décrivant le critère suivant lequel se base la spécialisation



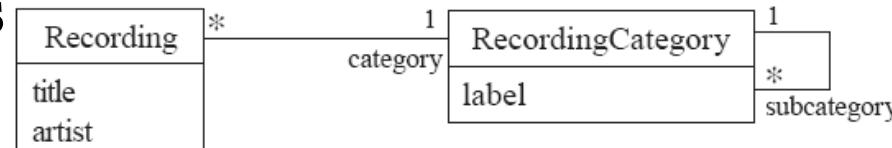
Éviter les généralisations inutiles

Exemple de hiérarchie
inappropriée

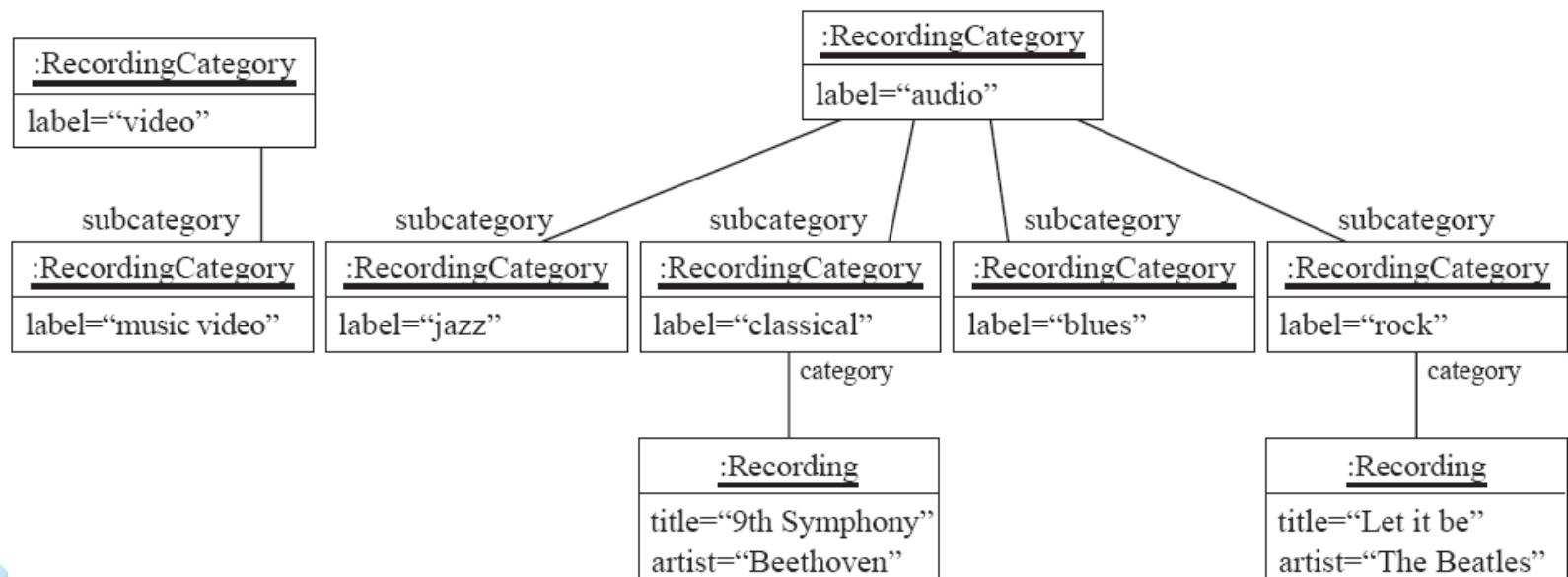


Éviter les généralisations inutiles (suite)

Un meilleur diagramme de classes et son diagramme d'instances



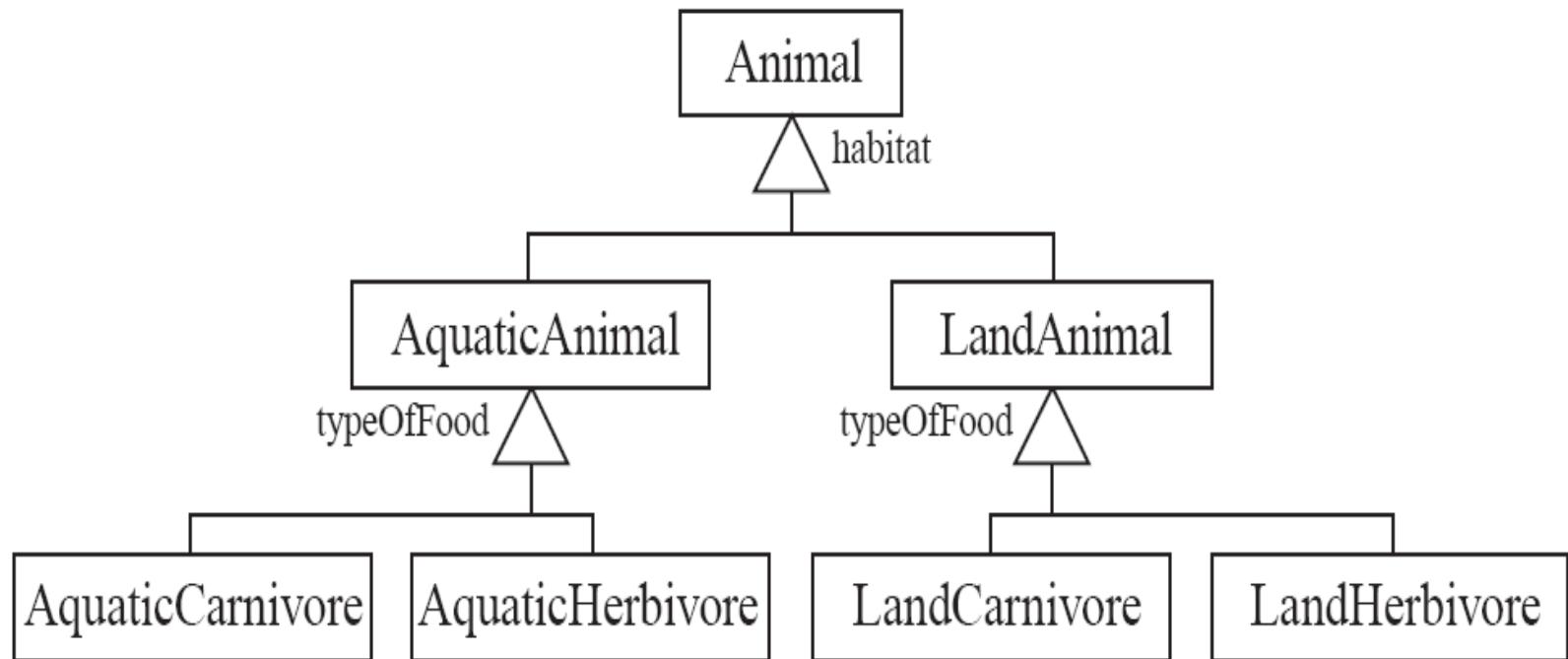
(a)



(b)

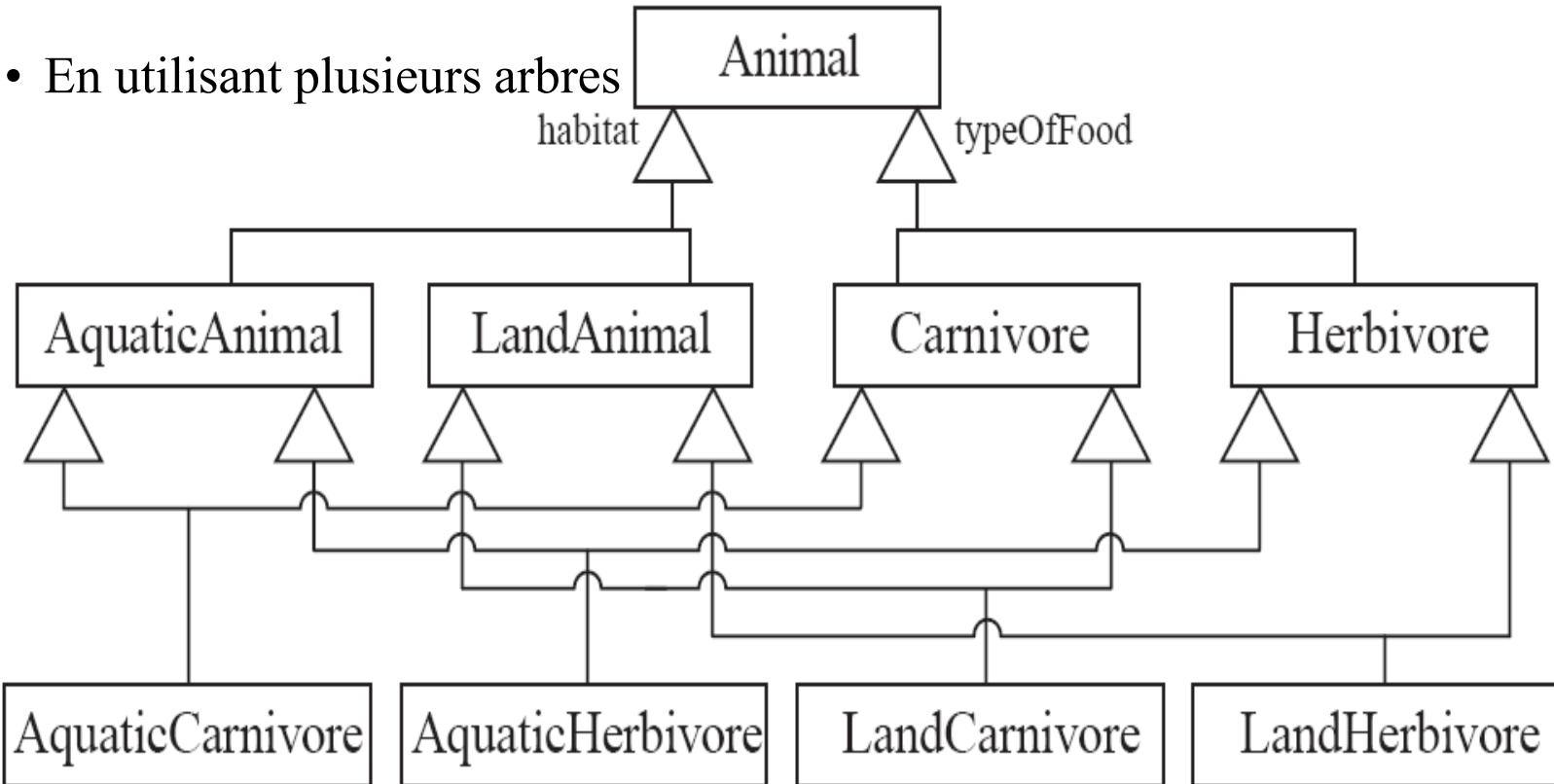
Hiérarchie à plusieurs discriminants

- En créant des généralisations à plusieurs niveaux



Hiérarchie à plusieurs discriminants

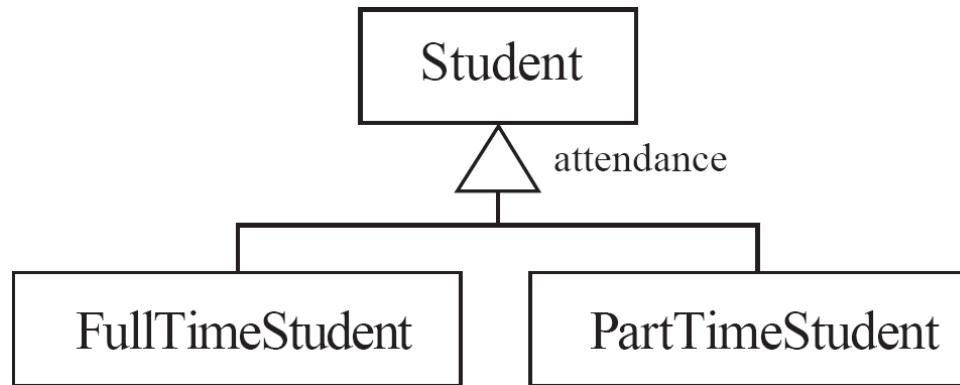
- En utilisant plusieurs arbres



- En utilisant le patron de conception Rôle-Acteur
(Chapitre 6)

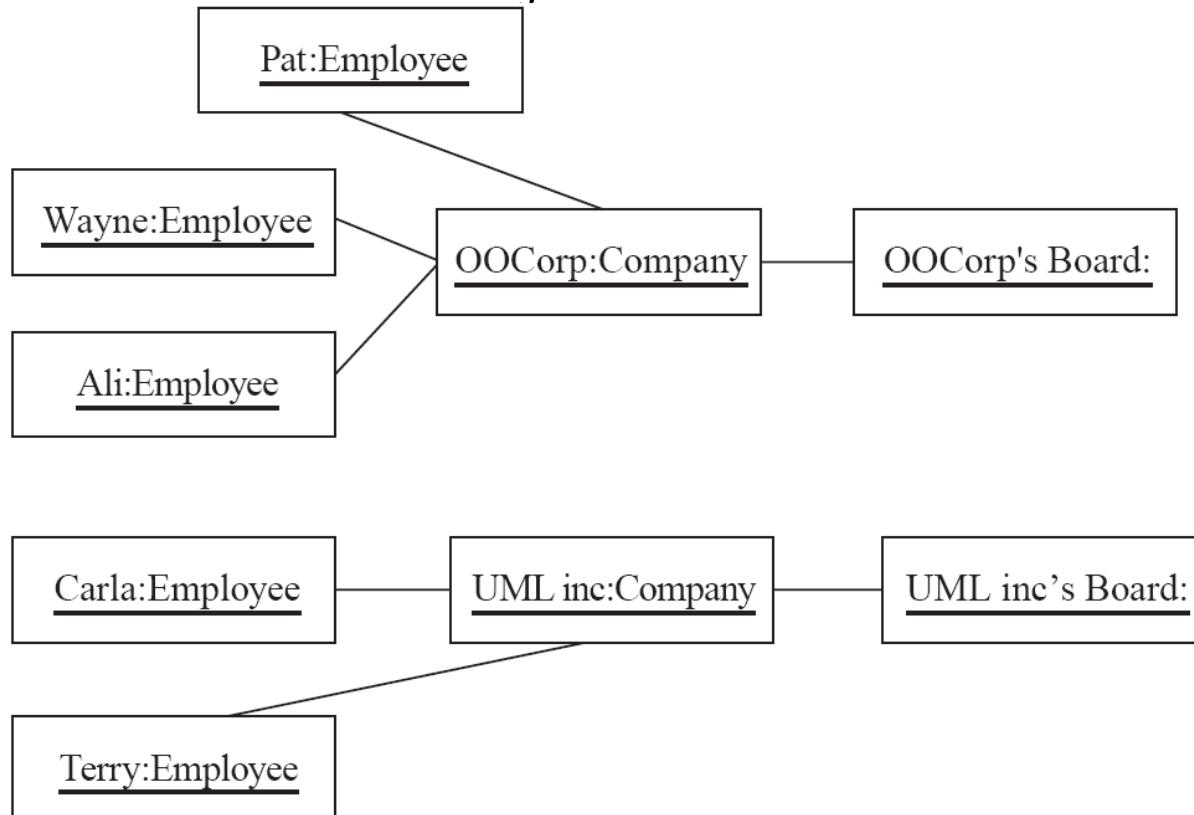
Éviter les changements de classes

- En fait, une instance ne peut jamais changer de classes



5.5 Diagrammes d'instance

- Un *lien* est l'instance d'une association
 - Tout comme un objet est l'instance d'une classe

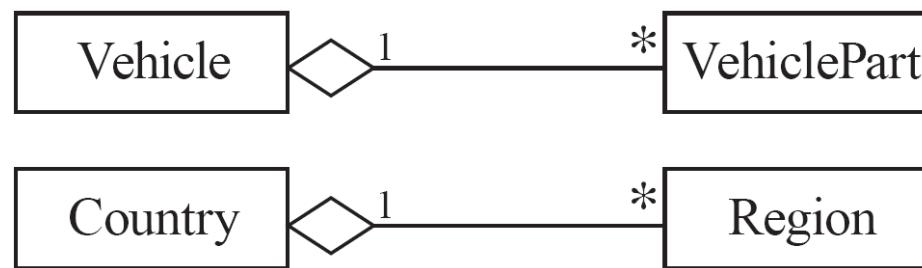


Associations versus généralisations dans un diagramme d'instances

- Une association décrit la relation qui existera entre deux *instances* en cours d'exécution.
 - Le diagramme d'instance montre les deux instances et le lien qui les unit
- Une généralisation décrit une relation entre classes dans un diagramme de classes
 - Les super-classes n'apparaissent pas dans un diagramme d'instances.
 - Une instance d'une classe est aussi une instance de sa super-classe

5.6 Notions avancées: Agrégation

- Une agrégation est une forme spéciale représentant une relation ‘partie-tout’.
 - Le ‘tout’ est souvent appelé l’ensemble ou l’agrégat
 - Ce symbole est une association de notation abrégée nommée fait-partie-De



A quel moment faut-il utiliser l'agrégation?

En général, une association peut être représentée comme une agrégation si:

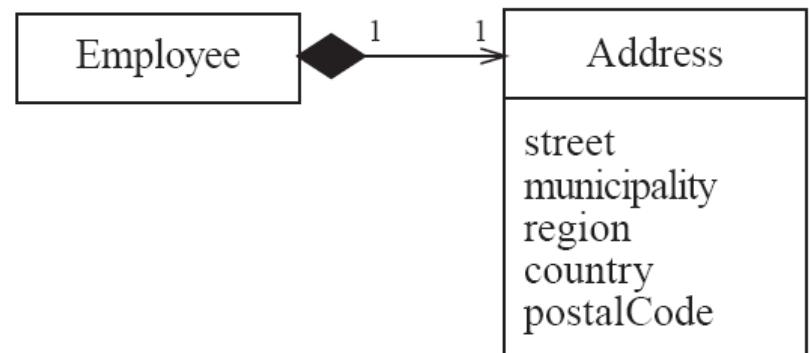
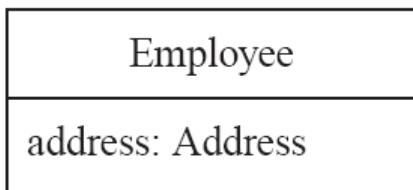
- il y a une relation de type *fait-partie-de*
- il y a une relation de type *est-composé-de*
- Lorsque quelque chose possède ou contrôle l'agréagat, il contrôle aussi ses parties

Composition

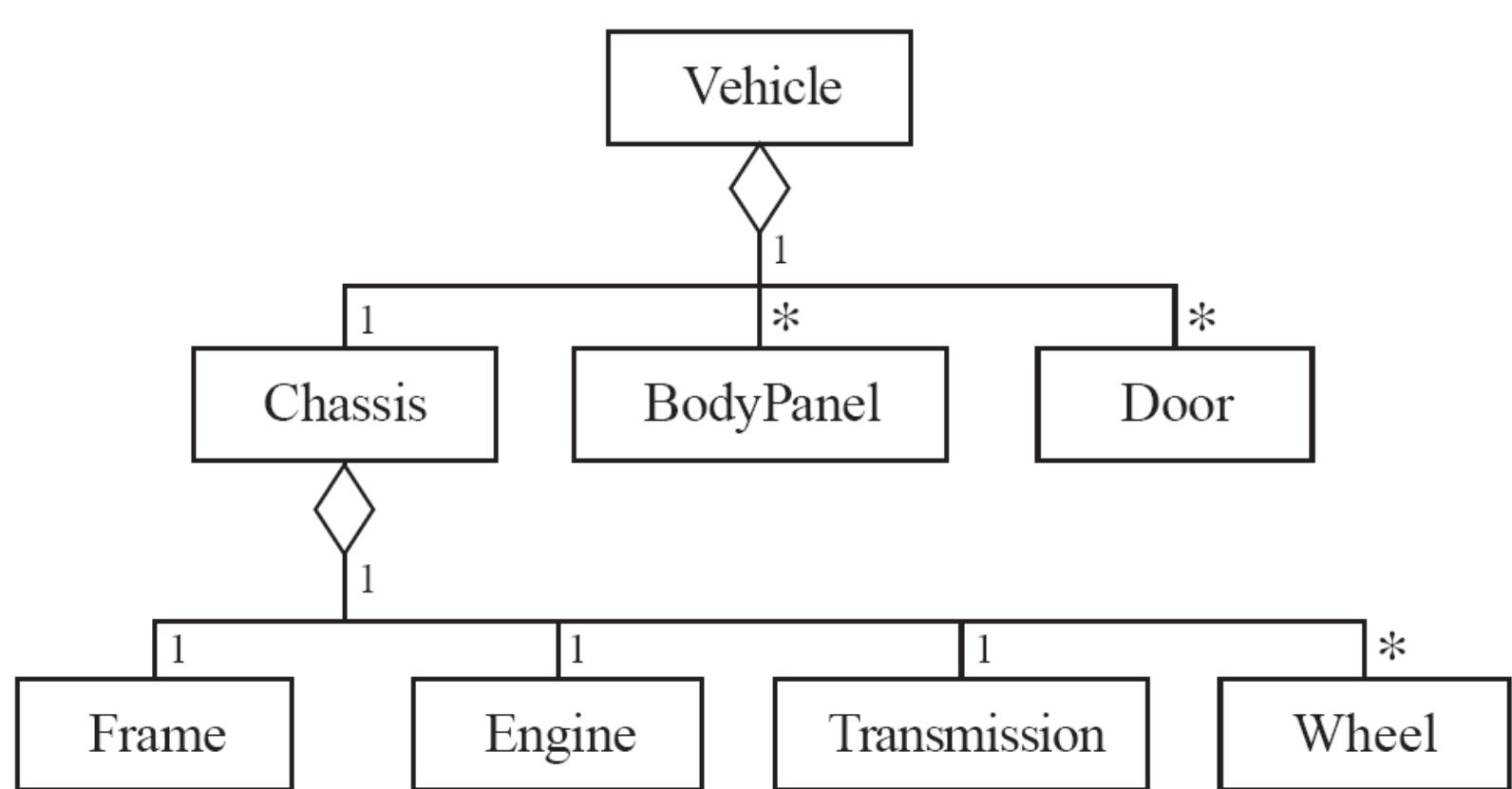
- Une *composition* est une forme forte d'agrégation
 - Si l'agrégat est détruit, alors ses parties le sont aussi



- Exemple: une adresse



Une hiérarchie d'agrégation



Propagation

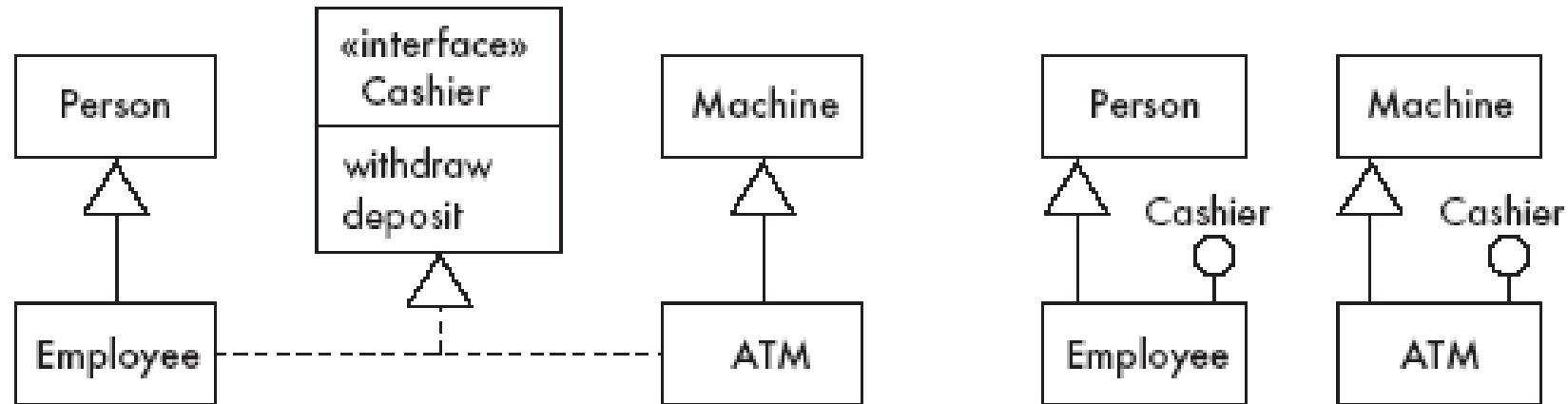
- La propagation est un mécanisme statuant que lorsqu'une opération est effectuée sur le tout elle doit aussi s'appliquer à ses parties
- La propagation permet aussi aux propriétés des parties de se propager vers le tout
- La propagation est à l'agrégation ce que l'héritage est à la généralisation.
 - La différence majeure est que:
 - L'héritage est un mécanisme implicite
 - La propagation doit être programmée



Les interfaces

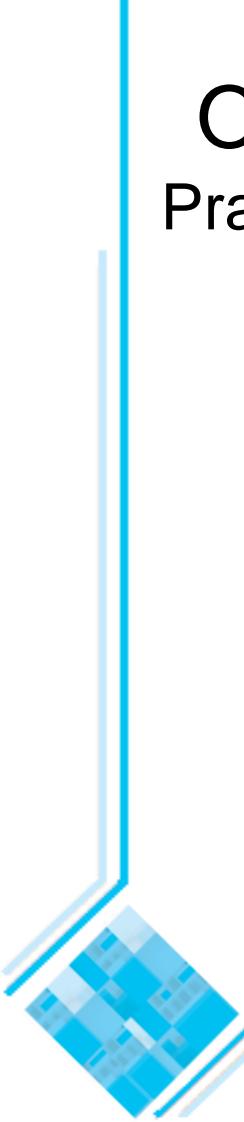
Une interface décrit une *portion du comportement visible d'un ensemble d'objets.*

- Une *interface* est similaire à une classe ne contenant que des opérations abstraites



Notes et descriptions

- **Il est toujours préférable d'inclure les diagrammes UML dans un document décrivant le système**
 - Les explications textuelles qui s'y trouve servent à donner des précisions, des justifications concernant ces diagrammes
- **Notes:**
 - Une note est une petite boite de texte incluses dans un diagramme UML
 - C'est l'équivalent d'un commentaire dans un programme



Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 5B:

Developing UML Class Diagrams

5.9 Le processus de développement des diagrammes de classes

Des diagrammes UML sont créés à différents moments et à différents niveaux de détails

- **Modèle exploratoire du domaine:**
 - Développé durant l'analyse de domaine dans le but d'en apprendre un peu plus sur le domaine
- **Modèle du domaine appartenant au système:**
 - Afin de modéliser certains aspects du domaine faisant partie du système
- **Modèle du système:**
 - Inclut les toutes les classes, y incluant les classes d'architecture des interfaces utilisateurs

Modèle du domaine-système vs modèle du système

- Le modèle du domaine-système omet plusieurs classes nécessaires à la conception du système complet
 - Contient souvent moins de la moitié des classes du système.
 - Devrait être indépendant:
 - des interfaces utilisateur
 - des classes d'architecture
- Le modèle du système inclut:
 - le modèle du domaine-système
 - les classes des interfaces utilisateur
 - les classes de l'architecture
 - les autres classes utilitaires

Séquence d'activités suggérée

- Identifier un premier ensemble de **classes candidates**
- Ajouter les **associations et attributs**
- Trouver les **généralisations**
- Lister les responsabilités principales de chacune des classes
- Décider quelles seront les **opérations**
- Effectuer quelques itérations jusqu'à satisfaction:
 - Ajouter ou retirer des classes, associations, attributs, généralisations, responsabilités ou opérations
 - Identifier les interfaces
 - Appliquer les patrons de conception appropriés (Chapitre 6)

Il ne faut être ni trop rigide ni trop désorganisé

Identifier les classes

- Lors du développement du modèle du domaine, les classes sont *découvertes*
- Lorsque le reste du système est élaboré, de nouvelles classes sont *inventées*
 - Ce sont les classes requises pour obtenir un système fonctionnel
 - L'invention de classes peut se produire à n'importe quel stage du développement
- La réutilisation doit toujours demeurer une priorité
 - Cadres d'application
 - Extensions du système
 - Systèmes similaires

Une technique simple pour découvrir les classes du domaines

- Examiner un document écrit telle qu'une description des exigences
- Extraire les *noms* et en faire des classes candidates
- Éliminer les noms qui:
 - sont redondants
 - représentent des instances
 - sont vagues ou trop généraux
 - sont inutiles dans le contexte de l'application
- Porter attention aux classes qui, dans le domaine, représentent des catégories d'utilisateurs.

Identifier les associations et les attributs

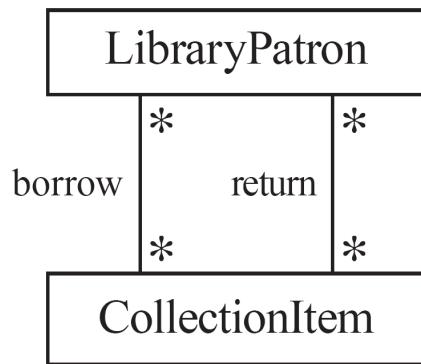
- Débuter avec les classes considérées centrales dans le système.
- Déterminer les données que chacune de ces classes devrait contenir et les relations avec les autres classes.
- Ajouter graduellement les autres classes.
- Éviter d'ajouter trop d'attributs ou d'associations à une classe
 - Un système manipulant peu d'information est toujours plus facile à maîtriser

Quelques trucs permettant d'identifier des associations

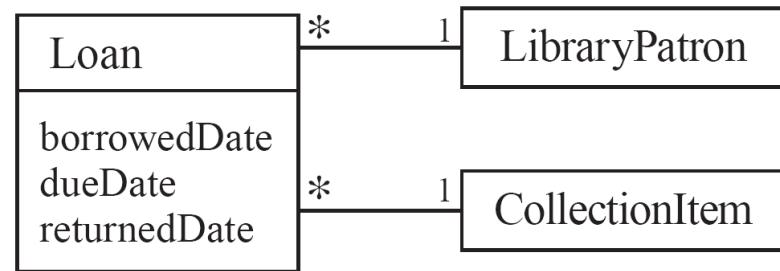
- Une association devrait exister si une classe
 - *possède*
 - *contrôle*
 - *est connecté à*
 - *est relié à*
 - *est une partie de*
 - *est fait de parties de*
 - *est membre de*
 - *a comme membres*
- Spécifier ensuite la multiplicité à chaque extrémité de l'association
- Étiquetter clairement cette association

Actions versus associations

- Une erreur fréquente est de représenter une *action* comme si elle était une association



Erreur, ici les associations sont en fait des actions



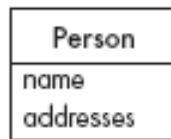
Préférable: L'opération d'emprunt crée un prêt

Identifier les attributs

- Déterminer l'information qui doit être maintenu dans chacune des classes
- Plusieurs noms ayant été rejetés comme classes deviennent alors des attributs
- Un attribut contient en général une valeur simple
 - E.g. chaîne de caractères, nombre

Quelques trucs permettant d'identifier des attributs

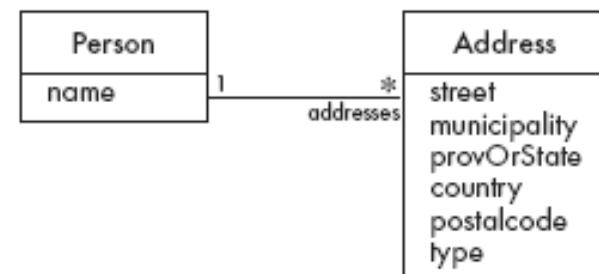
- Il ne doit pas y avoir duplication des attributs
- Si un sous-ensemble des attributs forme un groupe cohérent, alors une nouvelle classe devrait peut être introduite



Bad, due to
a plural attribute

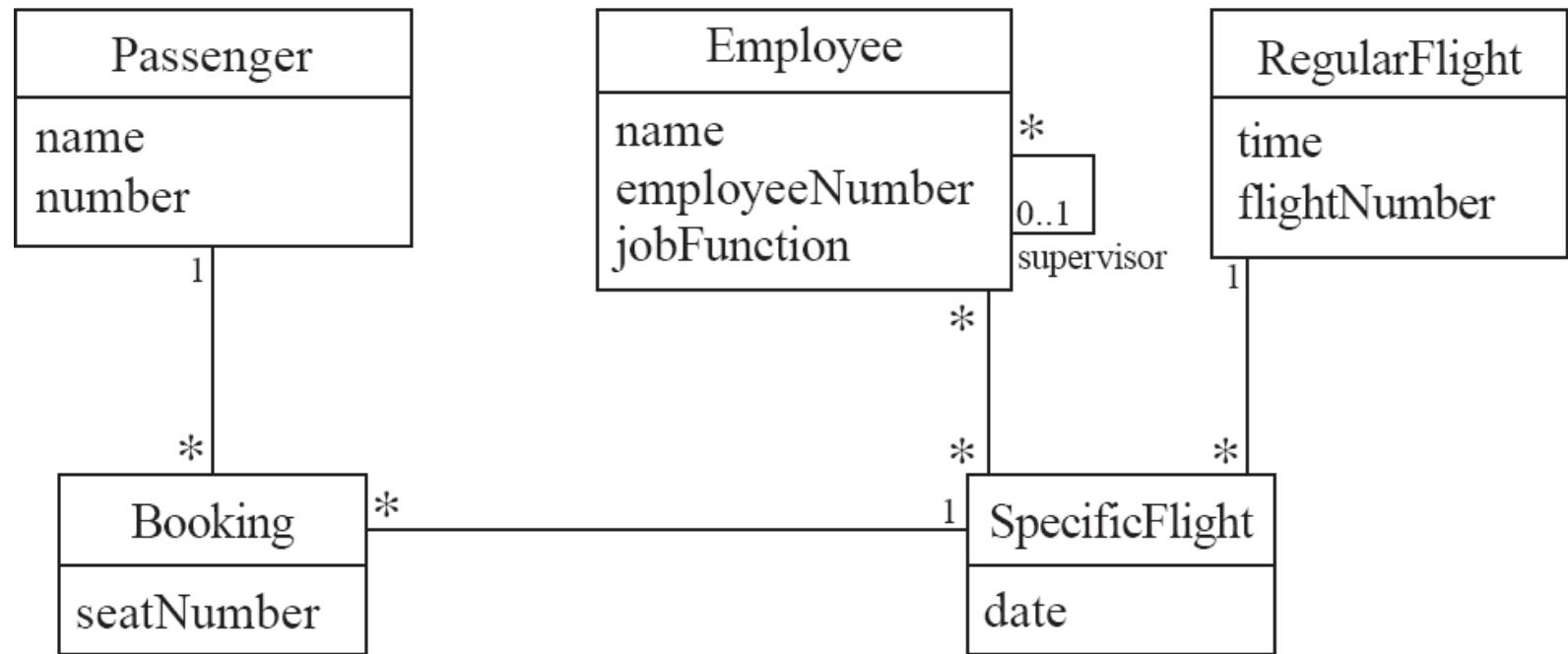


Bad, due to too many
attributes, and the
inability to add more
addresses



Good solution. The type indicates whether it
is a home address, business address etc.

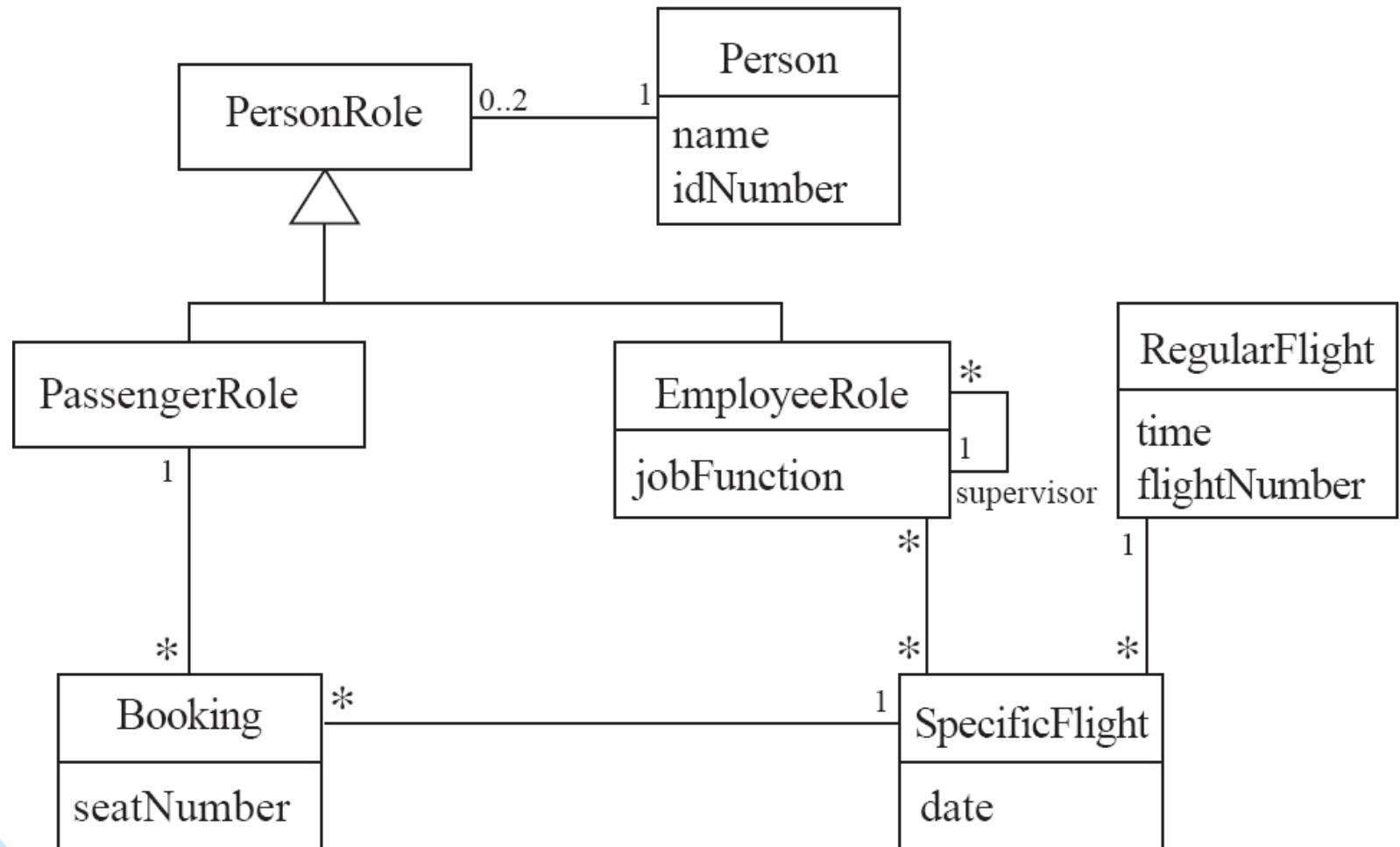
Un exemple (attributs et associations)



Identifier les généralisations et les interfaces

- Il existe deux façon d'identifier les généralisations:
 - De bas en haut
 - Grouper ensemble les classes similaires
 - De haut en bas
 - Spécialiser, au besoin, les classes plus générales
- Créer une *interface*, plutôt qu'une super-classe si
 - deux classes partageant certaines opérations sont considérées dissimilaires
 - une classe à généraliser possède déjà sa propre super-classe

Un exemple (généralisation)



Allouer des responsabilités aux classes

Une **responsabilité** est quelque chose que le système doit faire.

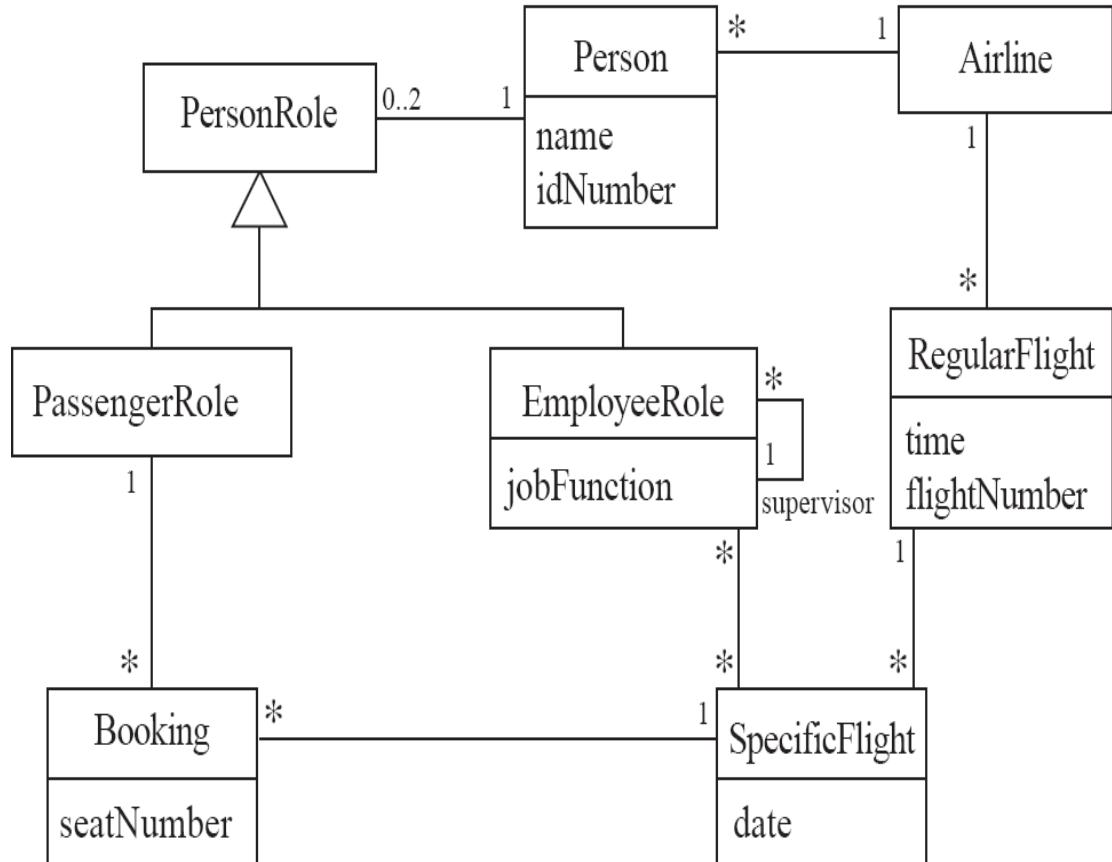
- Chacune des exigences fonctionnelles doivent être attribuées à l'une des classes
 - Toutes les responsabilités d'une classe doivent être reliées entre elles.
 - Si une classe a trop de responsabilités, elle devrait être scindée en plusieurs classes
 - Si une classe a aucune responsabilité, alors celle-ci est probablement inutile
 - Lorsqu'une responsabilité ne peut être attribuée à aucune classe, alors c'est qu'une nouvelle classe devrait être introduite
- Afin de déterminer les responsabilités
 - Effectuer une analyse de cas
 - Recherche des verbes et des noms décrivant des *actions*

Catégories de responsabilités

- Fixer et obtenir la valeur d'un attribut
- Créer et initialiser de nouvelles instances
- Sauvegarder et récupérer de l'information persistante
- Détruire des instances
- Ajouter et détruire des liens
- Copier, convertir, transformer, transmettre, afficher
- Calculer des résultats numériques
- Naviguer et rechercher
- Tout autre tâche...

Un exemple (responsabilités)

- Créer un nouveau vol régulier
- Rechercher un vol spécifique
- Modifier les attribut d'un vol
- Créer un vol spécifique
- Traiter la réservation d'un passager
- Annuler une réservation



Prototypage d'un diagramme de classes sur papier

- A mesure que les classes sont identifiée, écrire leur nom sur une petit carton
- Lorsqu'un attribut ou une responsabilité est identifiée, celle-ci est ajouter au carton de la classe correspondante
 - Si les responsabilités proposées ne peuvent entrer sur un seul carton:
 - C'est sans doutes qu'il faut scinder la classe en deux.
- Disposer les cartons sur un tableau afin de créer le diagramme de classes.
- Lier les cartons par des traits afin de représenter les associations et les généralisations.

Identifier les opérations

Les opérations servent à réaliser les responsabilités

- Il peut y avoir plusieurs opérations pour une responsabilité
- Les opérations de base réalisant ces responsabilités seront déclarées public
- Une méthode qui collabore à la réalisation d'une responsabilité sera, dans la mesure du possible, déclarée private

5.10 Implémenter un diagramme de classes en Java

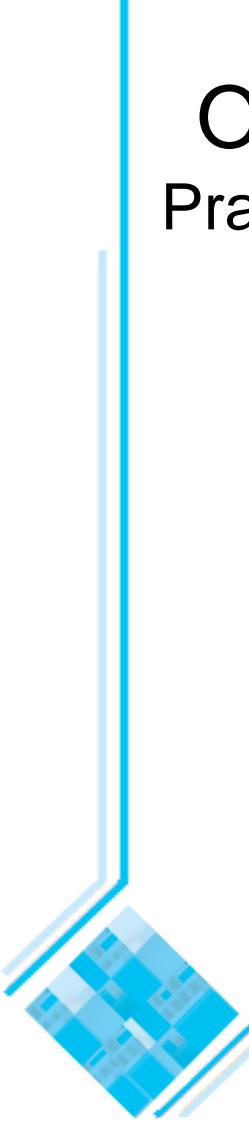
- Les attributs sont implementés en tant que variables d’instance
- Les généralisations sont implémentés en utilisant le mot-clé `extends`
- Les interfaces sont implémentés en utilisant le mot-clé `implements`
- Les associations sont normallement implémentées en utilisant des variables d’instance
 - Diviser chaque association en deux associations unidirectionnelles
 - Chacune des classe impliquées possède alors une variable d’instance
 - Pour une association où la multiplicité est de ‘un’ ou ‘optionnel’
 - Déclarer une variable référant à cette classe
 - Pour une association où la multiplicité est de ‘plusieurs’:
 - Utiliser une collection réalisant l’interfaces `List`, telle que `LinkedList`

Exemple: SpecificFlight

```
class SpecificFlight
{
    private Calendar date;
    private RegularFlight regularFlight;
    ...
    // Constructor that should only be called from
    // addSpecificFlight
    SpecificFlight( Calendar aDate, RegularFlight aRegularFlight)
    {
        date = aDate;
        regularFlight = aRegularFlight;
    }
}
```

Example: RegularFlight

```
class RegularFlight
{
    private List specificFlights;
    ...
    // Method that has primary responsibility
    public void addSpecificFlight(Calendar aDate)
    {
        SpecificFlight newSpecificFlight;
        newSpecificFlight = new SpecificFlight(aDate, this);
        specificFlights.add(newSpecificFlight);
    }
    ...
}
```



Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 5C:

Object Constraint Language

Object Constraint Language (OCL)

OCL est un langage permettant de décrire des contraintes présentes dans un module

- Une expression OCL décrit un énoncé logique (une contrainte) à propos du système et qui doit demeurer vraie
- Une contrainte ne doit pas amener d'effets secondaires
 - Elle n'effectue aucun calcul, ne modifie pas les données.
- Un énoncé OCL peut, par contre, spécifier quelles valeurs doivent avoir les attributs d'une classe ou d'une association

Object Constraint Language (OCL)

Les contraintes OCL spécifient des conditions qui doivent être respectées par le système modélisé.

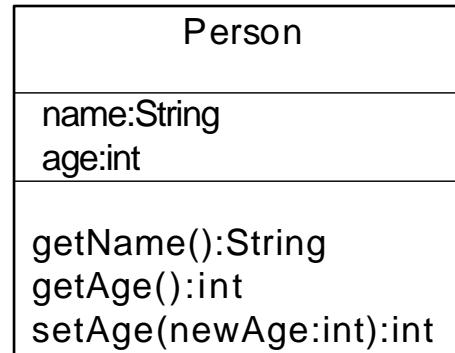
Quand utiliser OCL?

- Pour spécifier des invariants sur les classes et types d'un modèle UML
 - Un invariant est une contrainte qui devrait être vraie pour un objet pendant sa durée de vie complète.
- Pour spécifier de pré- et post-conditions sur les opérations

OCL – La notion de contexte

Une contrainte OCL est liée à un **contexte**, qui est le **type**, **l'opération** ou **l'attribut** auquel la contrainte se rapporte.

EX.

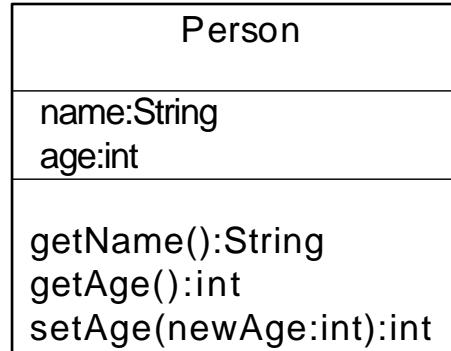


context Person inv: ...

context Person::setAge(a: int) ...

Navigation en OCL

1. L'accès (navigation) vers un **attribut** s'effectue en mentionnant l'attribut, derrière l'opérateur d'accès noté ‘**.**’.



EX.

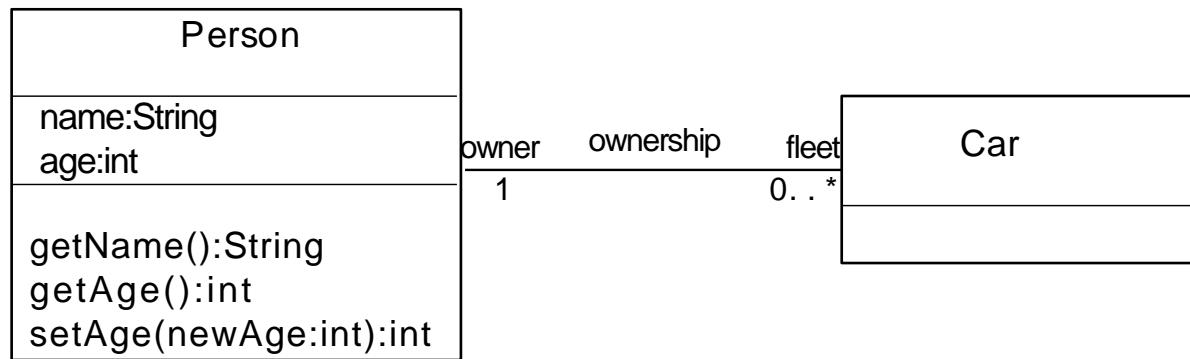
context Person

inv: self.age >=18;

Navigation en OCL

2. Navigation indirecte: La navigation le long des liens se fait en utilisant les **noms de rôles**

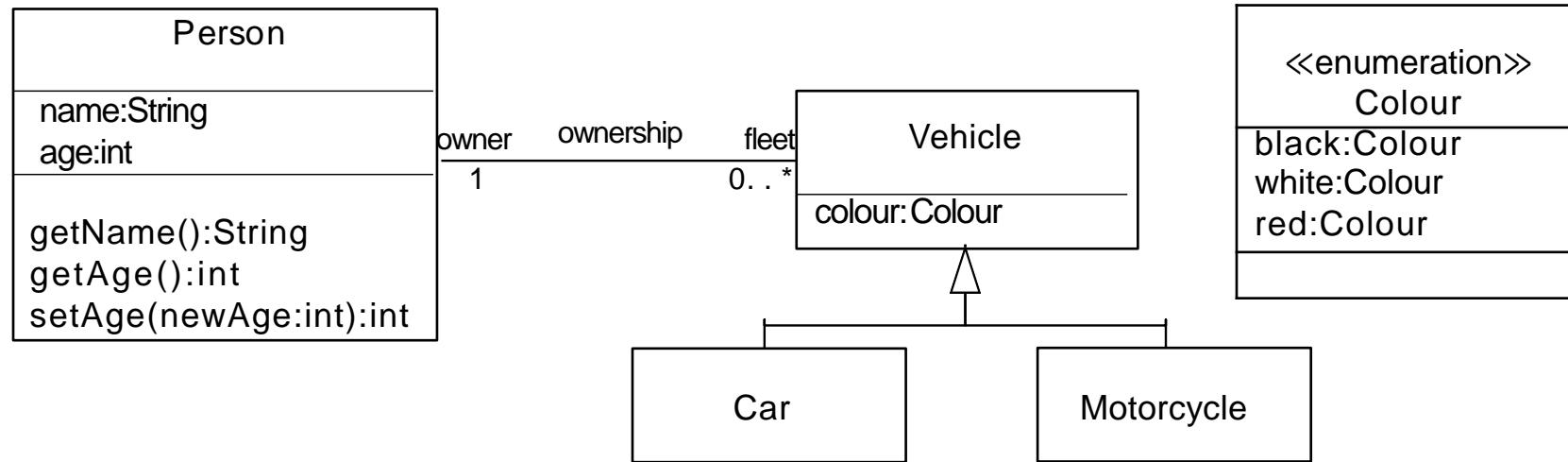
EX.



context Car

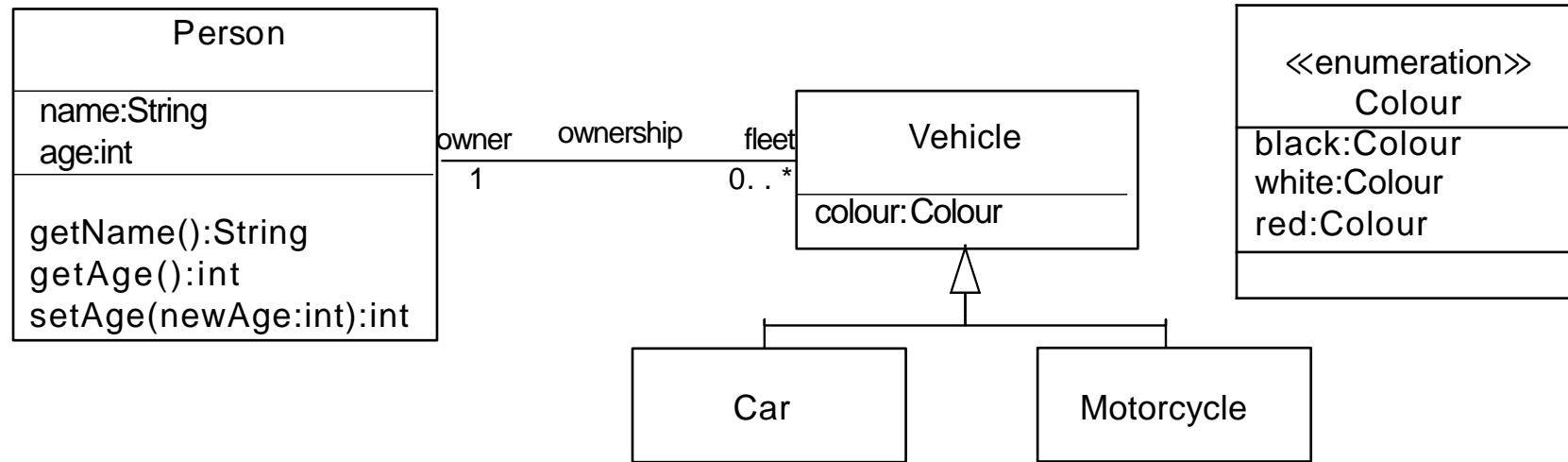
inv: self.owner.age >= 18;

Exemple: OCL - Invariant



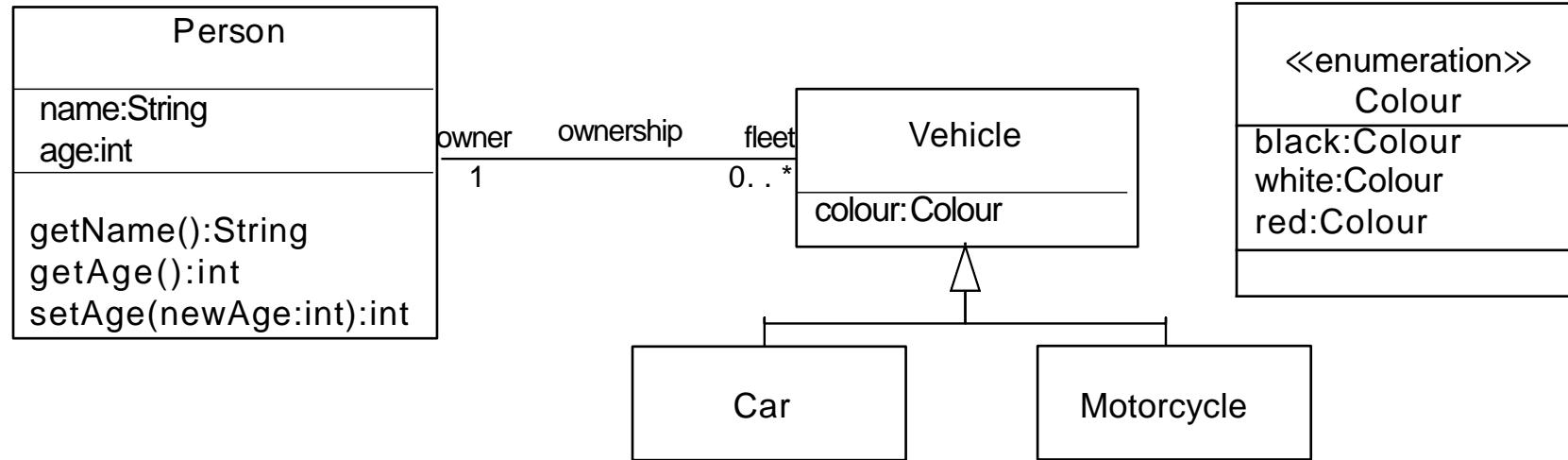
- Comment préciser qu'un propriétaire de voiture doit être au moins 18 ans?
- Comment exprimer qu'une personne peut posséder au plus trois voitures noires?
- Comment spécifier que la valeur de l'âge est x après avoir appelé `setAge(x)`?

Exemple: OCL - Invariant



- Comment préciser qu'un propriétaire de voiture doit être au moins 18 ans?
- Comment exprimer qu'une personne peut posséder au plus trois voitures noires?
- Comment spécifier que la valeur de l'âge est x après avoir appelé `setAge(x)`?

Exemple: OCL - Invariant

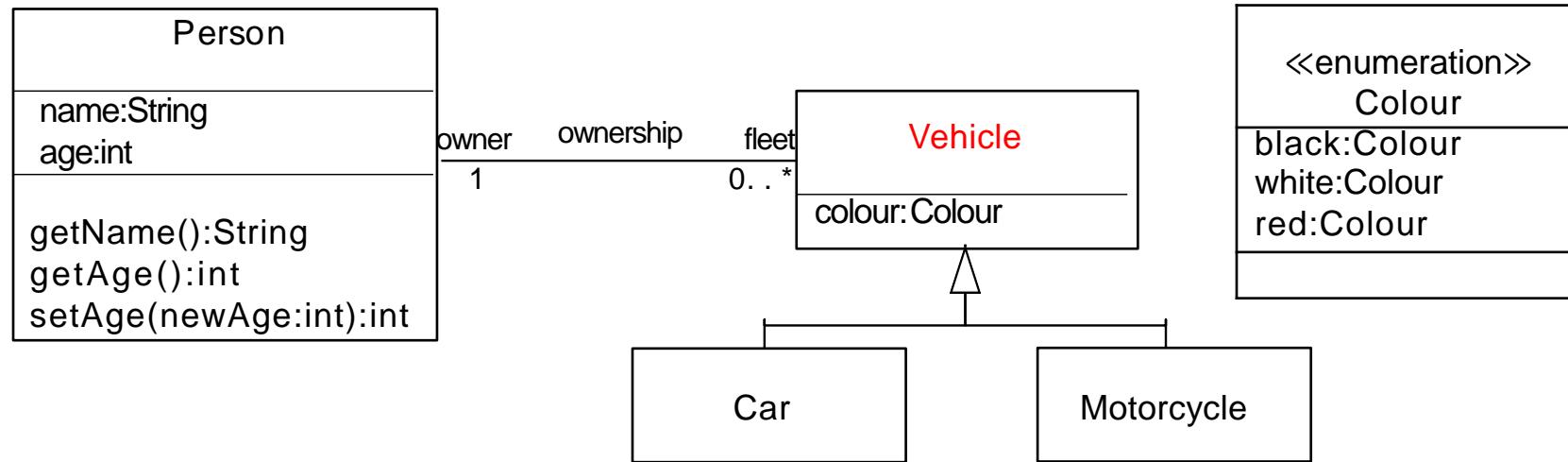


« Un propriétaire de véhicule doit avoir au moins 18 ans »:

context Vehicle

inv: self. owner. age >= 18

Exemple: OCL - Invariant

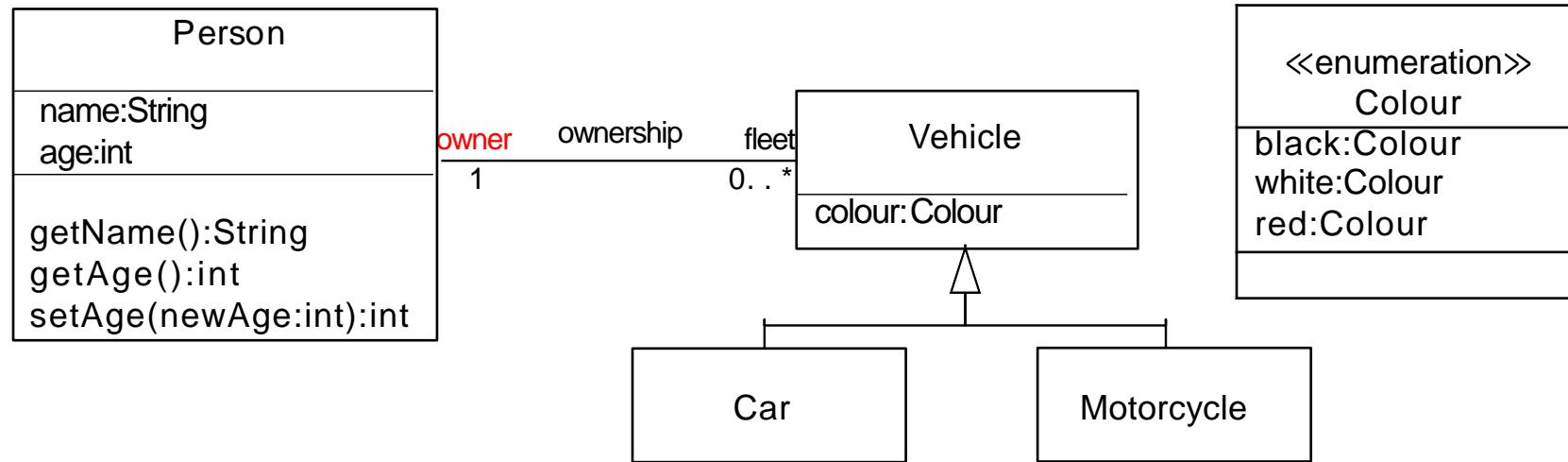


« Un propriétaire de véhicule doit avoir au moins 18 ans »:

context Vehicle

inv: self. owner. age >= 18

Exemple: OCL - Invariant

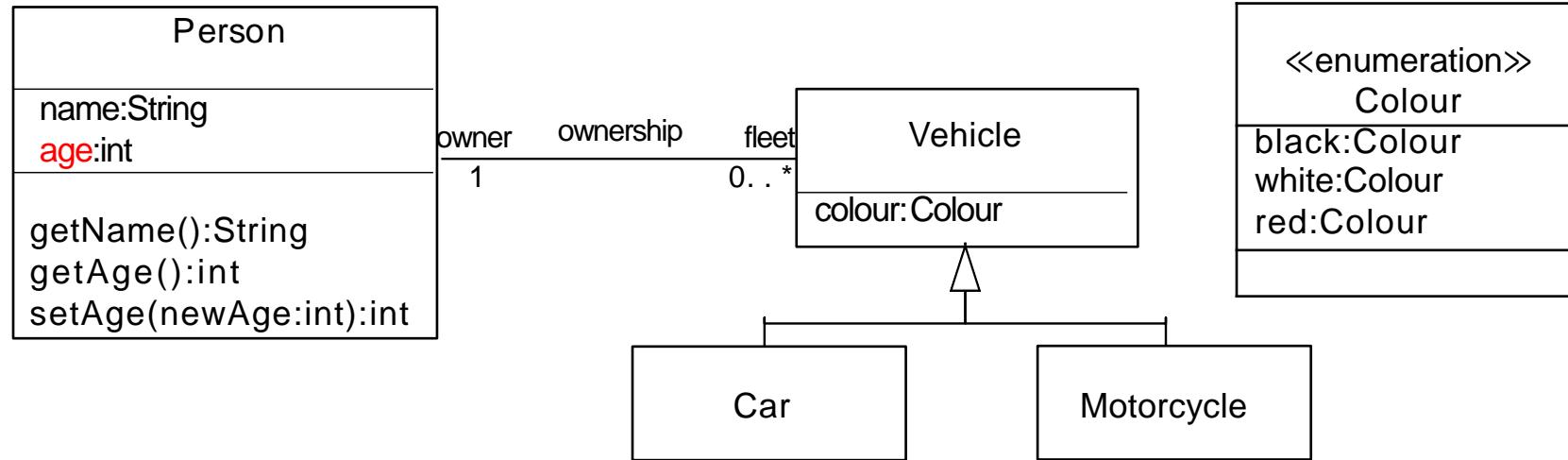


« Un propriétaire de véhicule doit avoir au moins 18 ans »:

context Vehicle

inv: self. owner. age >= 18

Exemple: OCL - Invariant

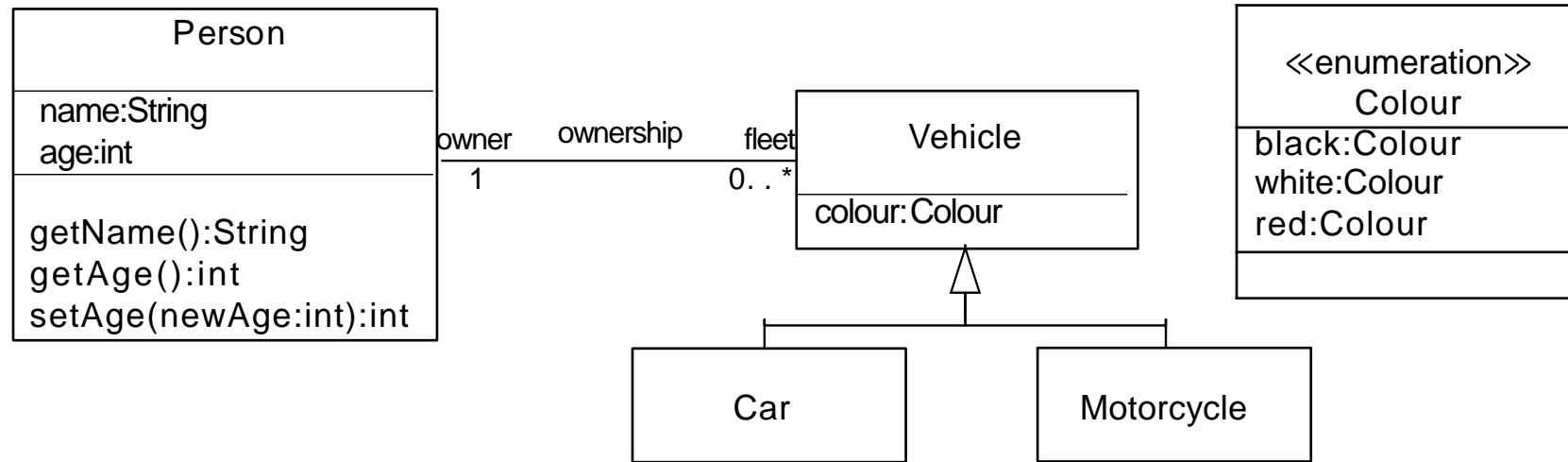


« Un propriétaire de véhicule doit avoir au moins 18 ans »:

context Vehicle

inv: self. owner. age >= 18

Exemple: OCL - Invariant

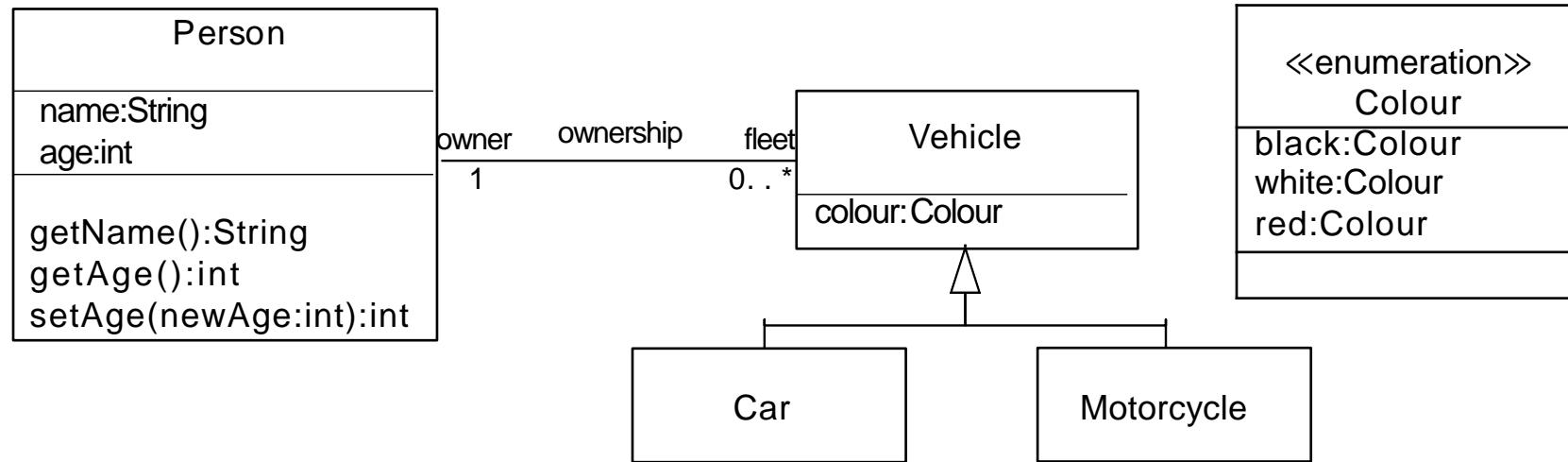


Qu'est-ce que ça veut dire?

context Person

inv : self.age >= 18

Exemple: OCL - Invariant



Une personne doit être au moins 18 ans

context Person

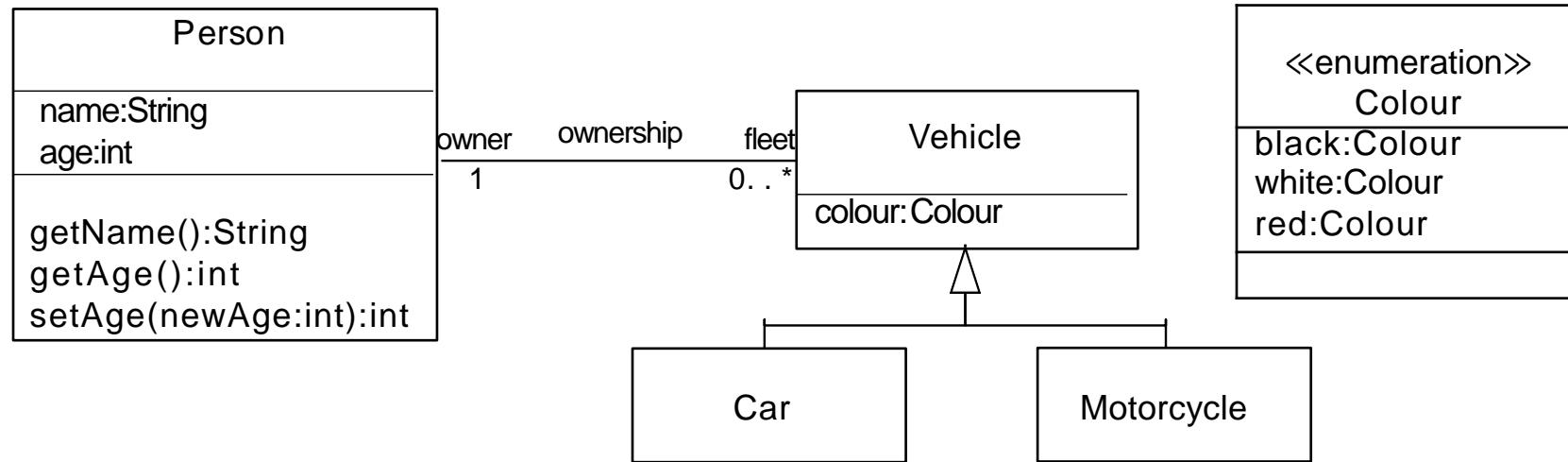
inv: self.age >= 18

Une personne qui est propriétaire de véhicule doit être au moins 18 ans

context Vehicle

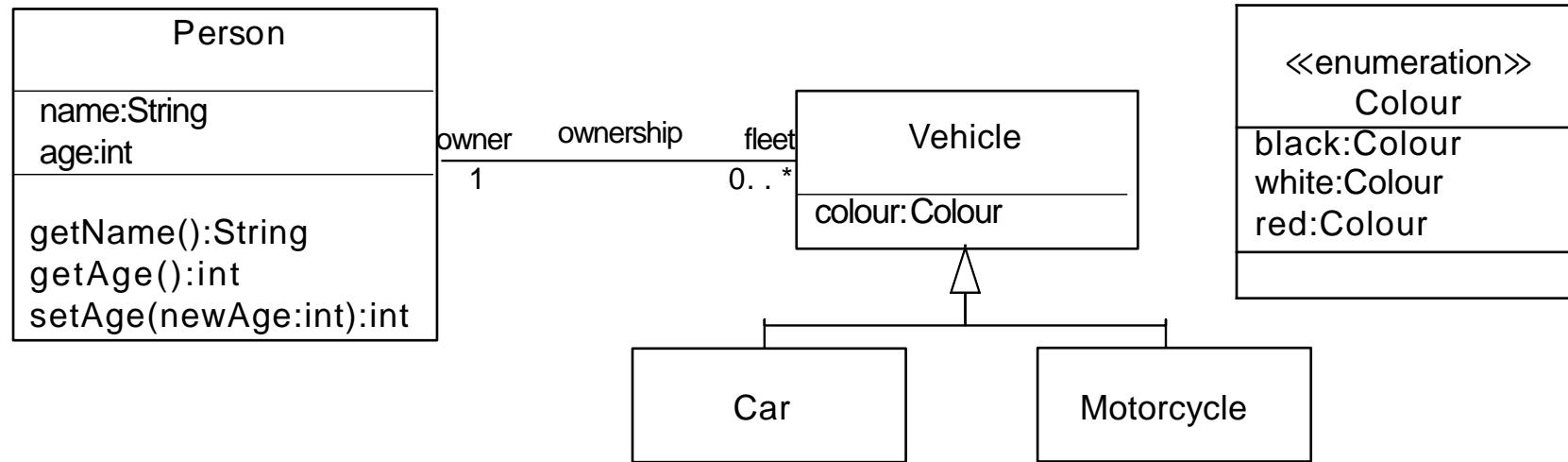
inv : self. owner. age >= 18

Exemple: OCL - Invariant



- Comment préciser qu'un propriétaire de voiture doit être au moins 18 ans?
- Comment exprimer qu'une personne peut posséder au plus trois voiture noire?
- Comment spécifier que la valeur de l'âge est x après avoir appelé `setAge(x)`?

Exemple: OCL - Invariant

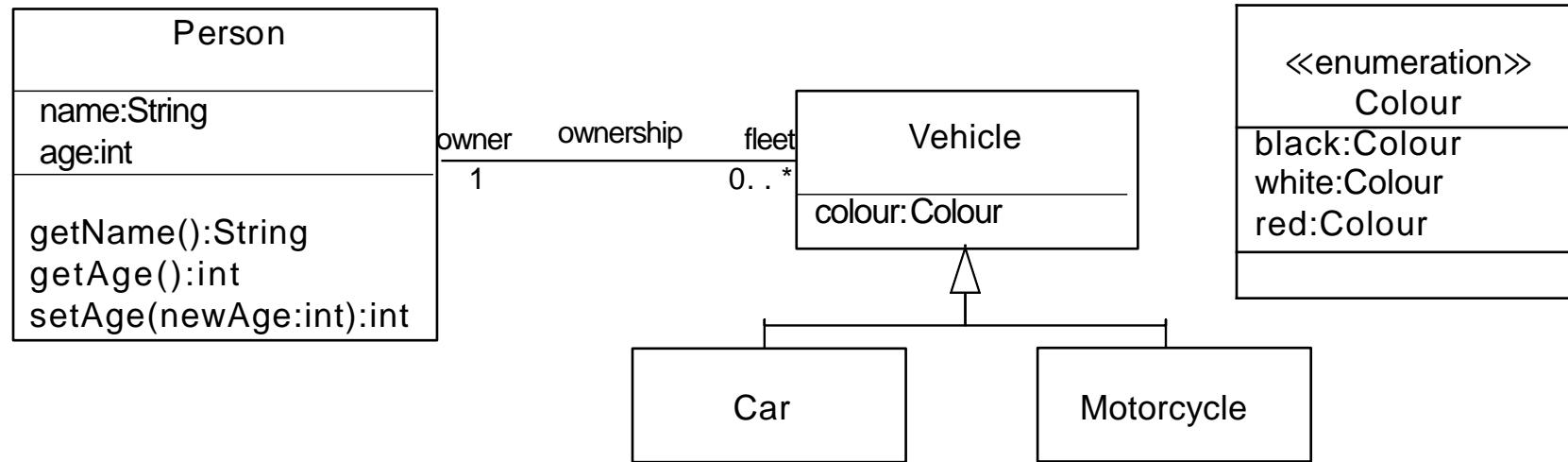


« Personne ne peut possède plus de 3 véhicules »:

context Person

inv: self.fleet-> size() <= 3

Exemple: OCL - Invariant



« Personne ne possède plus de 3 véhicules noirs »:

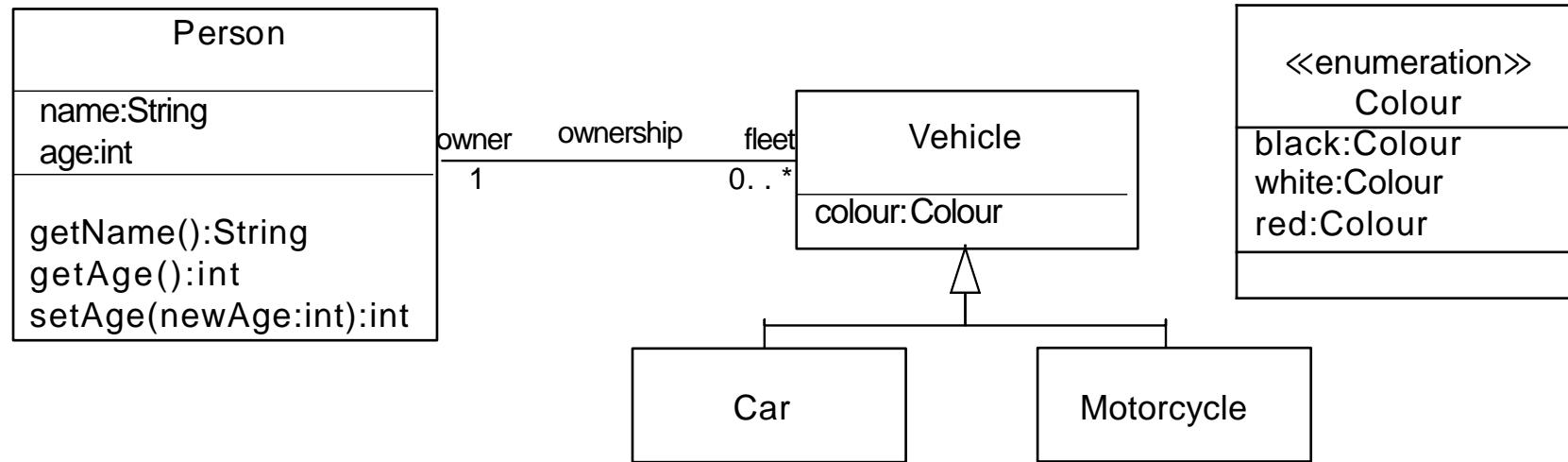
context Person

inv: self.fleet->select(v | v.colour = Colour.black)->size() <= 3

Quelques opérations sur les collections

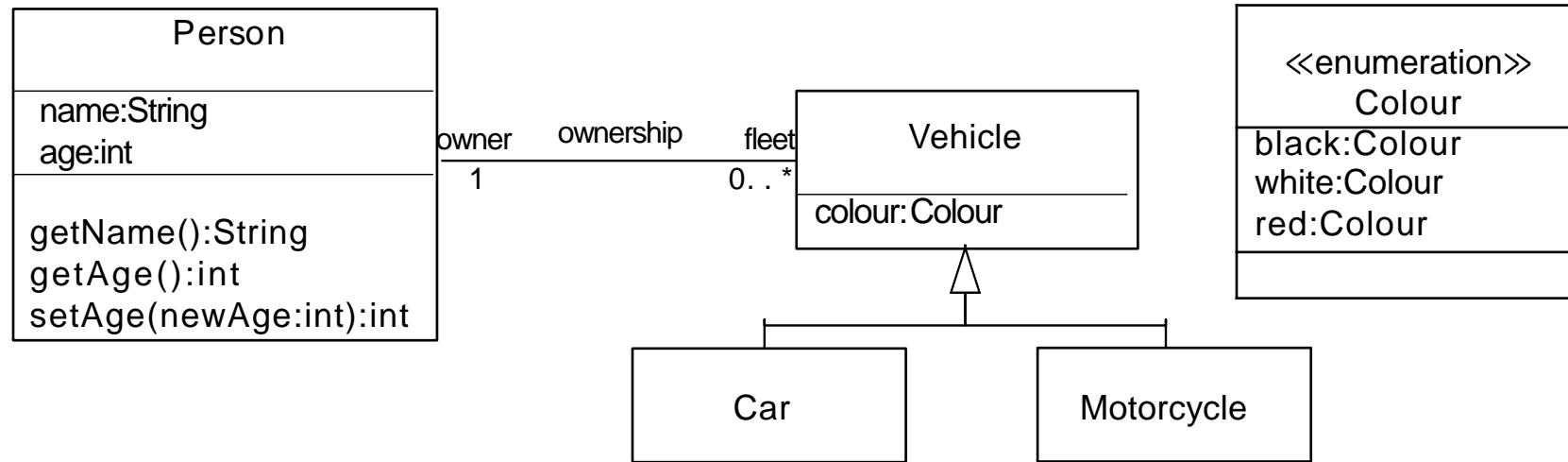
Opération	Usage
isEmpty	aCollection->isEmpty()
notEmpty	aCollection->notEmpty()
includes	aCollection->includes(anObject)
excludes	aCollection->excludes(anObject)
sum	aCollectionOfNumbers->sum()
exists	aCollection->exists(booleanExpression)
size	aCollection->size()

Exemple: OCL - Invariant



- Comment préciser qu'un propriétaire de voiture doit être au moins 18 ans?
- Comment exprimer qu'une personne peut posséder au plus trois voitures noires?
- **Comment spécifier que la valeur de l'âge est x après avoir appelé `setAge(x)`?**

Exemple: OCL – Pré et Post Condition



context Person::setAge(newAge: int):int

pre: self.age \geq 0 and newAge \geq 0

post: self.age = newAge

Exemple: OCL – Pré et Post Condition

```
context Person::setAge(newAge: int):int  
    pre: self.age ≥ 0 and newAge ≥ 0  
    post: self.age = newAge
```

```
int setAge (int newAge) {  
    if (age≥0 && newAge≥0) {this.age = newAge; }  
    return this.age;  
}
```



Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 6:

Using Design Patterns

6.1 Introduction aux Patrons de conception

Les aspects récurrents d'un design sont appelés *patrons de conception*.

- Un *patron* est l'esquisse d'une solution réutilisable à un problème général rencontré dans un contexte particulier
- Plusieurs de ces patrons ont été documentés et mis à la disposition des développeurs afin qu'il puissent en faire usage.
- Un bon patron de conception devrait
 - Être aussi général que possible
 - Contenir une solution qui a été démontrée efficace pour résoudre un problème précis dans un contexte identifié.

L'étude des patrons de conception est une façon efficace d'apprendre à partir de l'expérience des autres

Description d'un patron de conception

Contexte:

- La situation générale ou le patron s'applique

Problème:

—Une ou deux courtes phrases résumant la difficulté principale.

Forces:

- Les points principaux à considérer dans la résolution du problème

Solution:

- La manière recommandée de résoudre le problème dans un contexte donné

Anti-patrons: (Optionnel)

- Solutions de moindre qualité ou inadéquate dans le contexte donné

Patrons apparentés: (Optionnel)

- Patrons similaire au patron proposé.

Références:

- Le concepteur du patron ou celui qui l'a inspiré.

6.2 Le patron Abstraction-Occurrence

- ***Contexte:***

- Souvent, dans le modèle du domaine, il existe un ensemble d'objets reliés entre eux (*occurrences*).
- Les membres d'un tel ensemble partagent des informations communes
 - Bien qu'ils diffèrent aussi.

- ***Problème:***

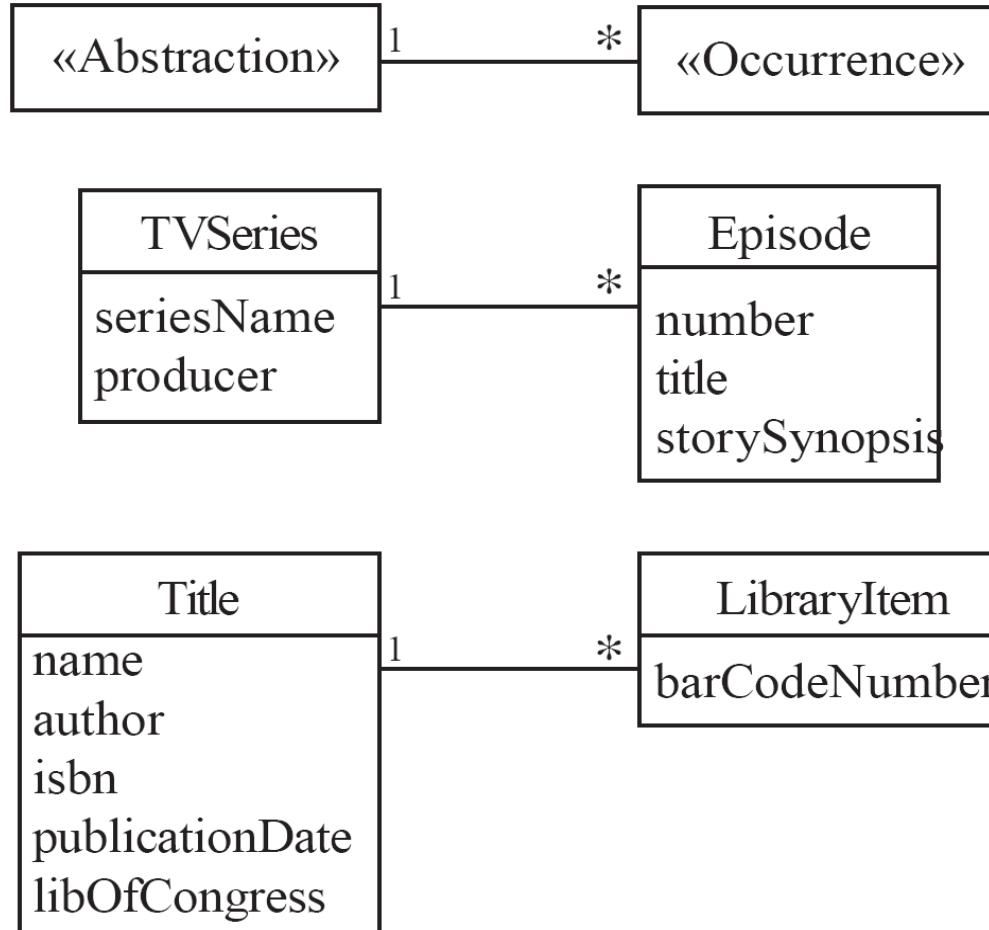
- Quelle est la meilleure façon de représenter un tel ensemble d'occurrences dans un diagramme de classes?

- ***Forces:***

- Vous souhaitez représenter les propriétés de chaque membre d'ensemble d'occurrences sans avoir à dupliquer l'information commune

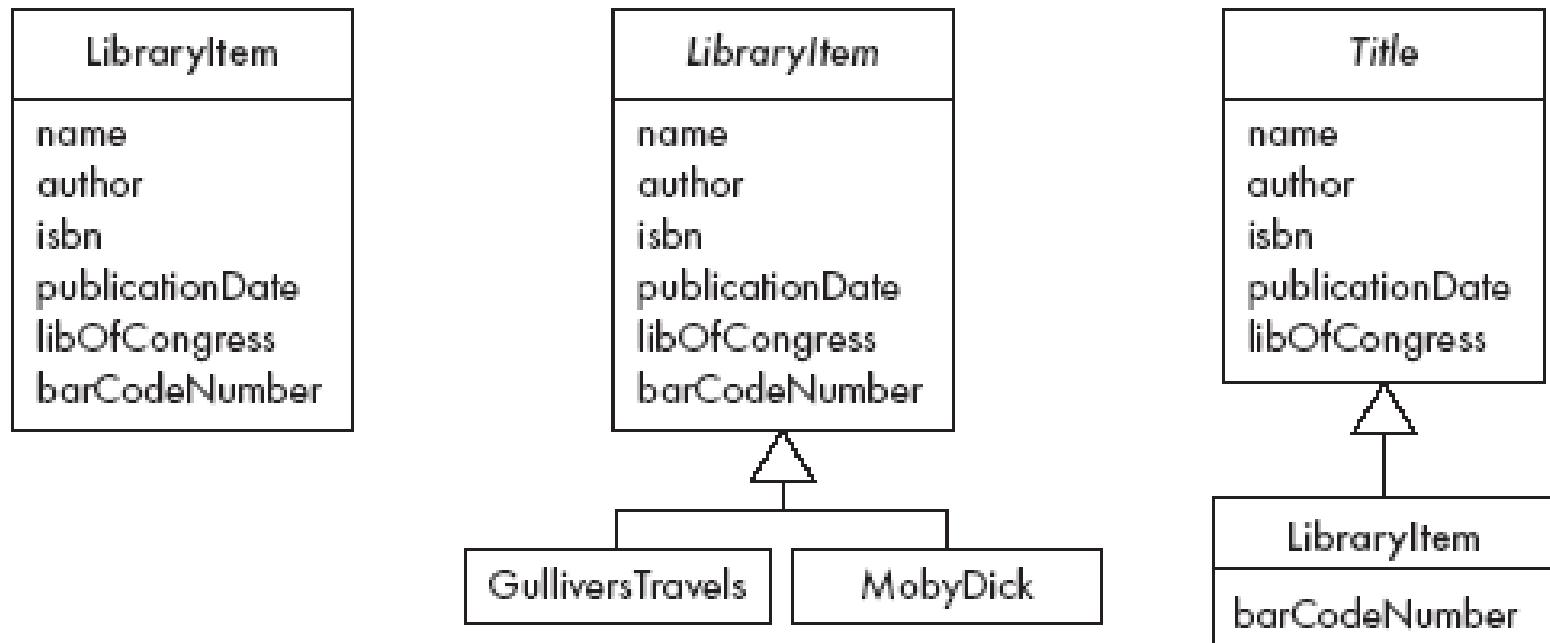
Abstraction-Occurrence

- *Solution:*



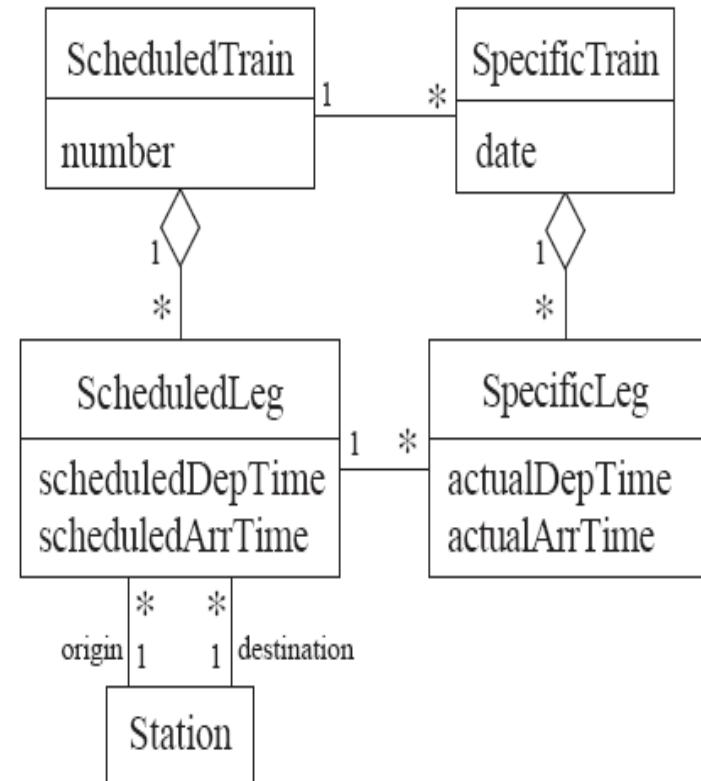
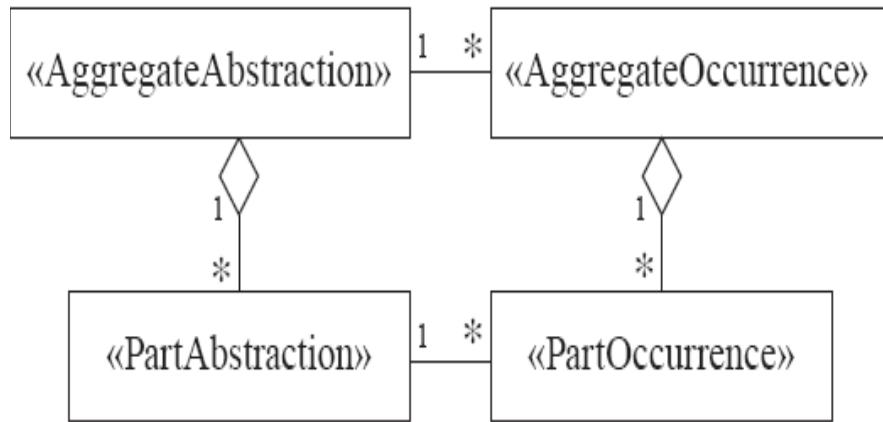
Abstraction-Occurrence

Anti-patrons:



Abstraction-Occurrence

Variante en carré

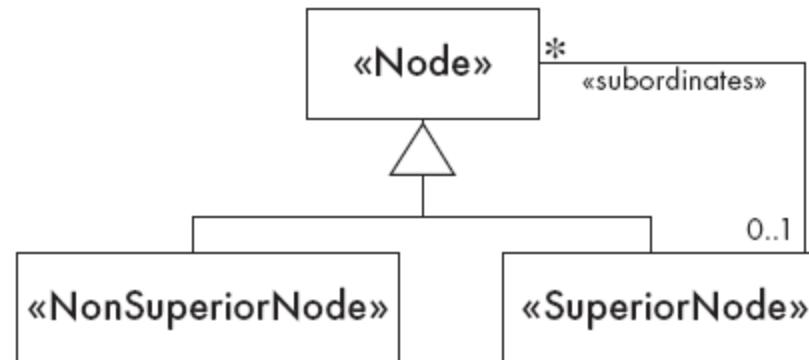


6.3 Le patron de Hiérarchie généralisée

- ***Contexte:***
 - Les objets d'une hiérarchie peuvent avoir un ou plusieurs objets au-dessus d'eux (leurs supérieurs),
 - Et un ou plusieurs objets sous eux (leurs subordonnés).
 - Certains objets ne peuvent avoir de subordonnés
- ***Problème:***
 - Comment représenter une telle hiérarchie d'objets dans laquelle certains objets ne peuvent avoir de subordonnés?
- ***Forces:***
 - Vous recherchez une solution flexible afin de représenter une hiérarchie générale
 - Les objets partagent plusieurs propriétés et comportements communs

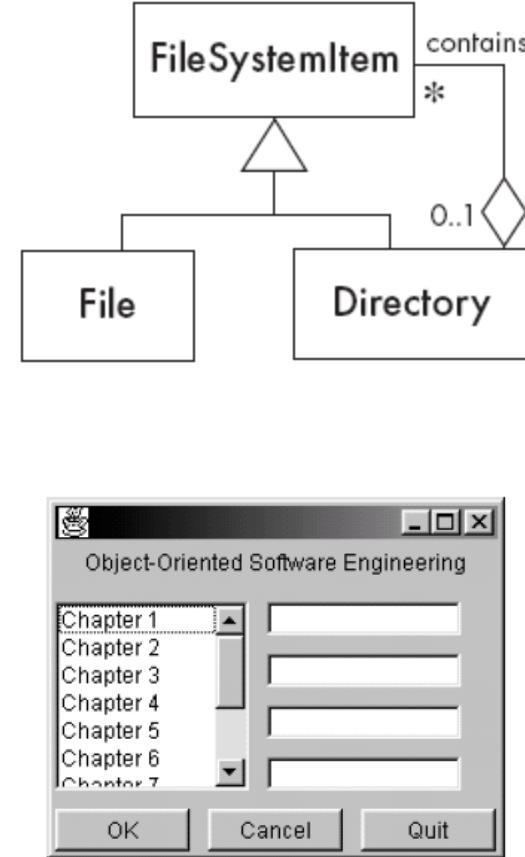
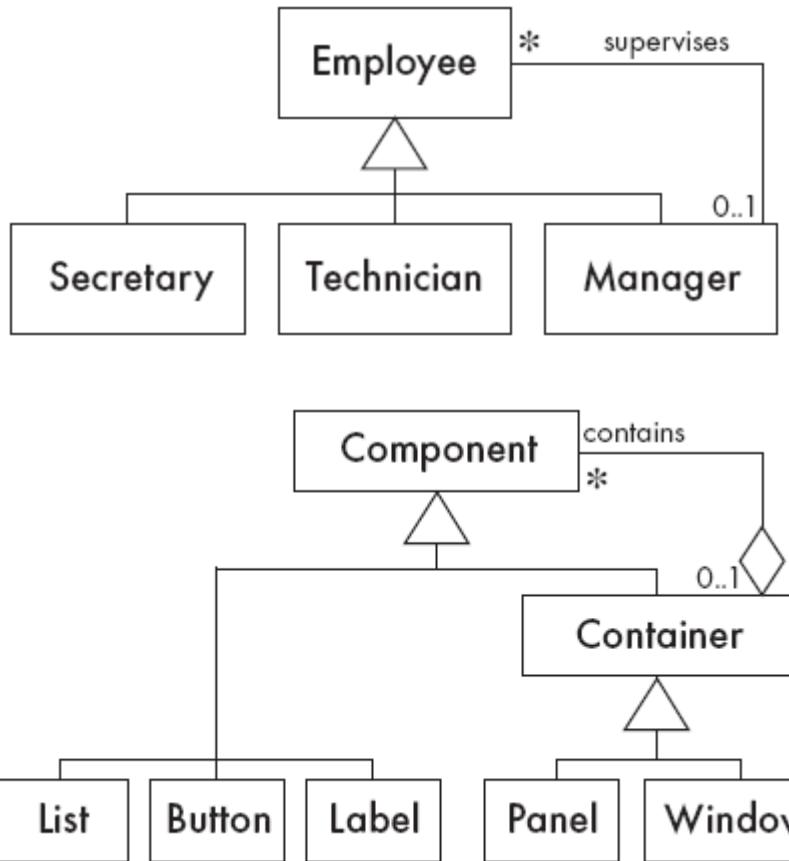
Hiérarchie généralisée

- *Solution:*



Hiérarchie généralisée

- *Solution:*



6.4 Le patron Acteur-Rôle

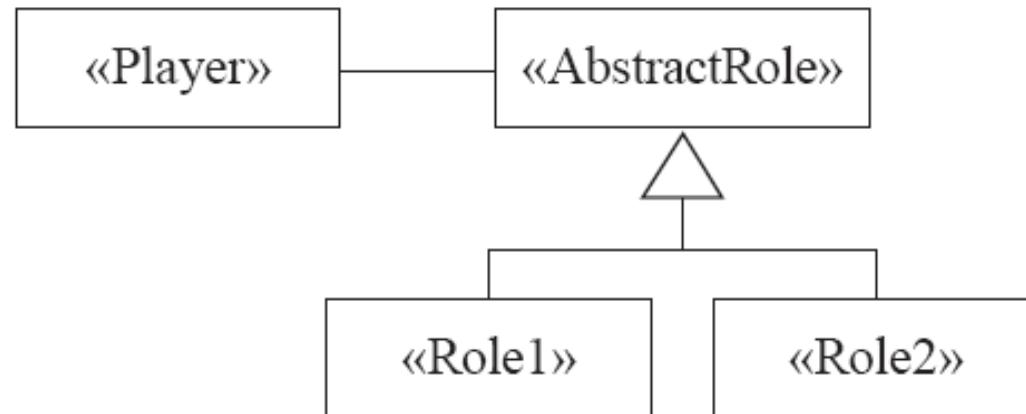
- ***Contexte:***
 - Un *rôle* correspond à un ensemble de propriétés particulières associées à un objet dans un contexte particulier.
 - Un objet peut jouer plusieurs rôles dans différents contextes.
- ***Problème:***
 - Comment modélisé une situation où un objet peut jouer plusieurs rôles (conjointement ou consécutivement)?

Acteur-Rôle

- ***Forces:***

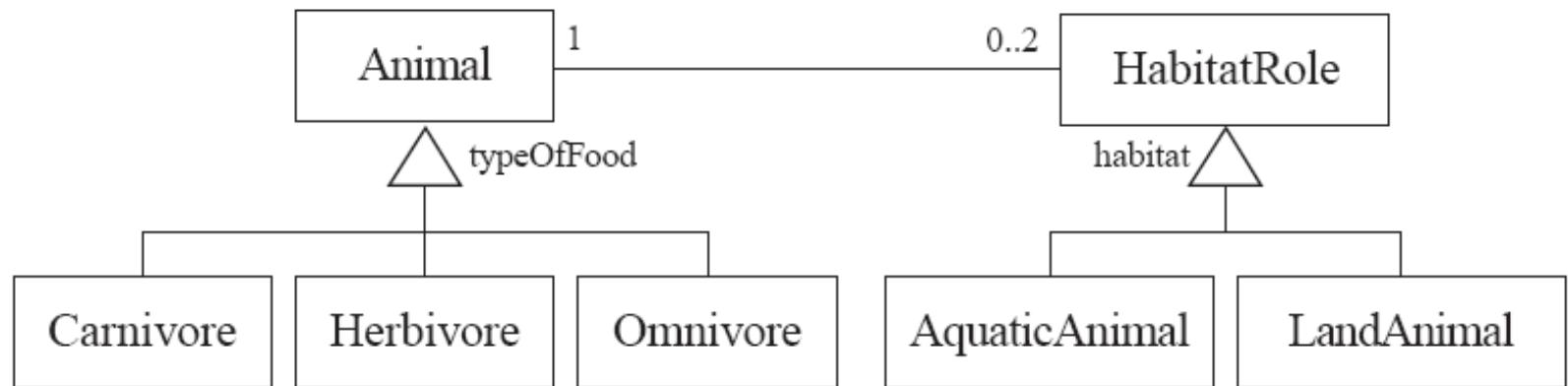
- Il est souhaitable de renforcer l'encapsulation en intégrant l'information associée à chaque rôle dans des classes distinctes.
- Il faut éviter l'héritage multiple.
- Un objet ne peut changer de classe d'appartenance

- ***Solution:***



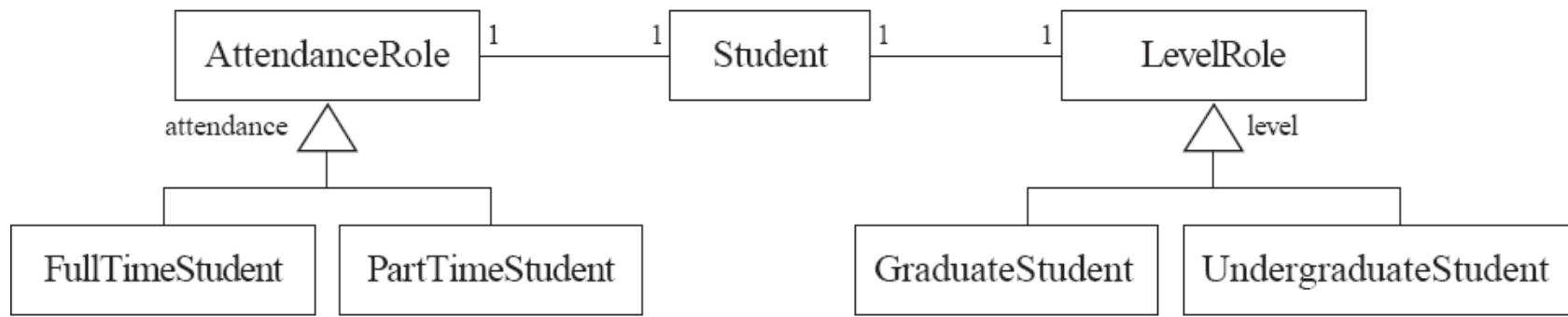
Acteur-Rôle

Exemple 1:



Acteur-Rôle

Exemple 2:



Acteur-Rôle

Anti-patrons:

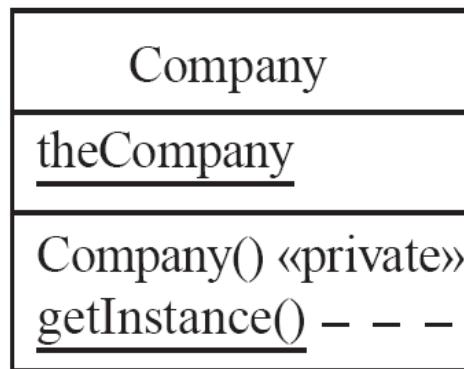
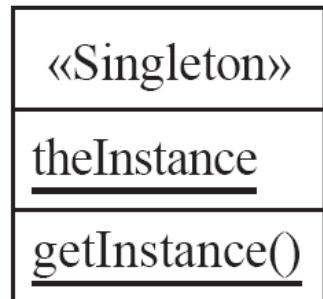
- Fusionner tous les comportements et propriétés dans une seule classes.
- Créer des rôles étant sous-classes de la classe acteur.

6.5 Le patron Singleton

- ***Contexte:***
 - Il est fréquent de retrouver des classes pour lesquelles il ne doit exister qu'une seule instance (*singleton*)
- ***Problème:***
 - Comment assurer qu'il ne sera jamais possible de créer plus d'une instance de cette classe?
- ***Forces:***
 - L'utilisation d'un constructeur public ne peut pas garantir qu'au plus une seule instance sera créée.
 - L'instance du singleton doit être accessible de toutes les classes qui en ont besoin

Singleton

- ***Solution:***



```
if (theCompany==null)  
    theCompany= new Company();
```

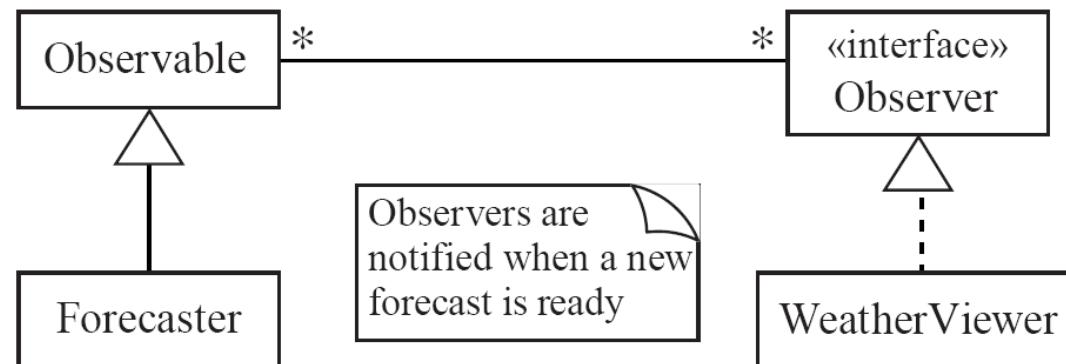
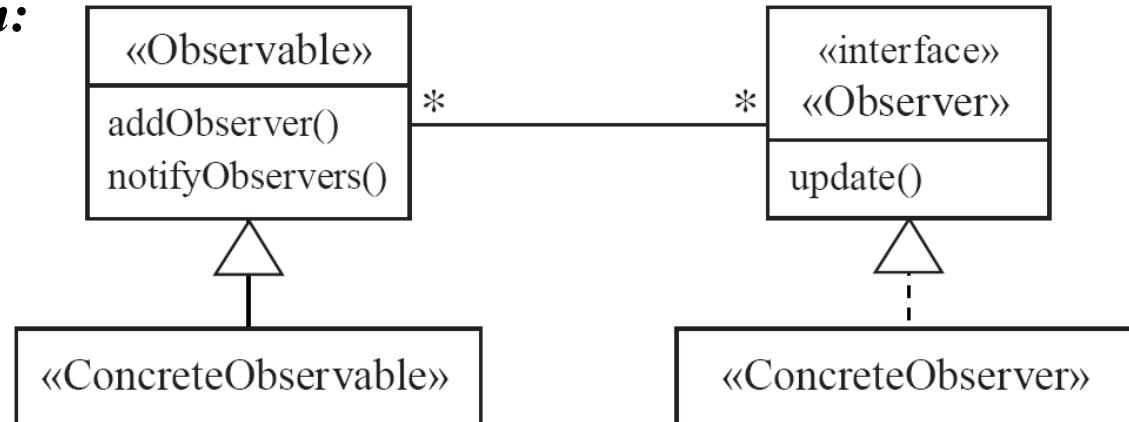
```
return theCompany;
```

6.6 The patron Observateur

- ***Contexte:***
 - Quand une association est créée entre deux classes, celles-ci deviennent inséparables.
 - Si vous souhaitez réutilisée l'une de ces classes, la seconde doit aussi être réutilisée.
- ***Problème:***
 - Comment réduire l'interconnexion entre classes, en particulier si elle appartiennent à des modules ou des sous-systèmes différents?
- ***Forces:***
 - Vous souhaitez maximiser la flexibilité du système

Observateur

- **Solution:**



Observateur

Anti-patrons:

- Connecter un observateur directement à un observé de façon telle que chacun contient une référence à l'autre.
- Faire des observateurs une sous-classe des observés.

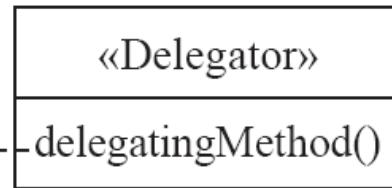
6.7 Le patron Délégation

- ***Contexte:***
 - Vous devez concevoir une méthode dans une classe
 - Vous réalisez qu'une autre classe possède une méthode qui fournit le service requis
 - L'héritage n'est pas approprié
- ***Problème:***
 - Comment réutiliser une méthode existant dans une autre classe?
- ***Forces:***
 - Vous souhaitez minimiser le temps de développement par la réutilisation

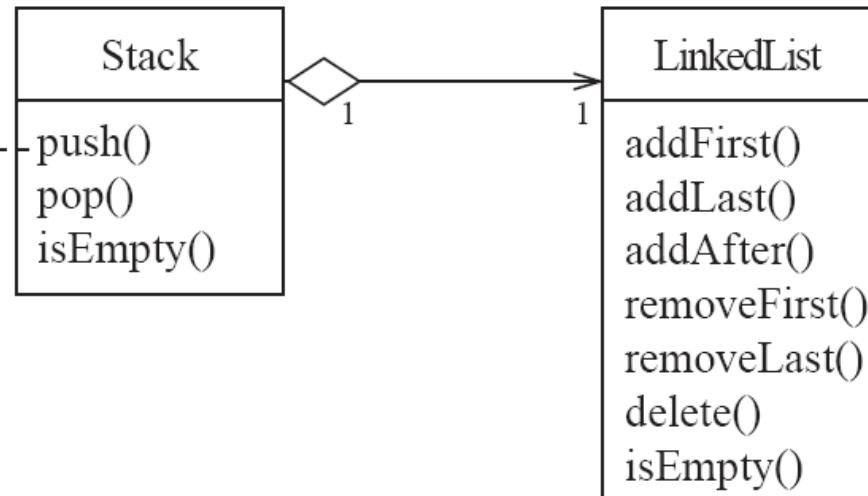
Délégation

- ***Solution:***

```
delegatingMethod()  
{  
    delegate.method();  
}
```

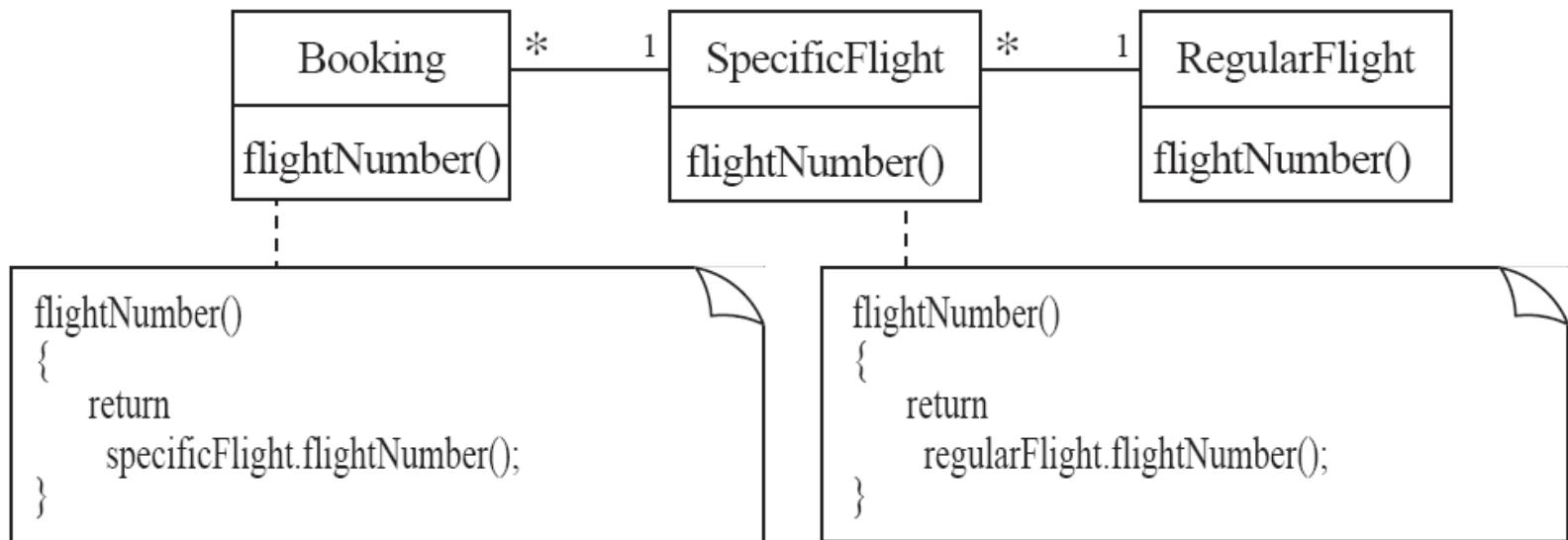


```
push()  
{  
    list.addFirst();  
}
```



Délégation

Exemple:



Délégation

Anti-patrons

- Utiliser la généralisation et ainsi hériter de la méthode ainsi que de toutes les autres
- Au lieu d'avoir une seule méthode dans le délégué appelant la méthode du délégué, avoir plusieurs méthodes appelant la méthode du délégué
- Accéder à des classes lointaines

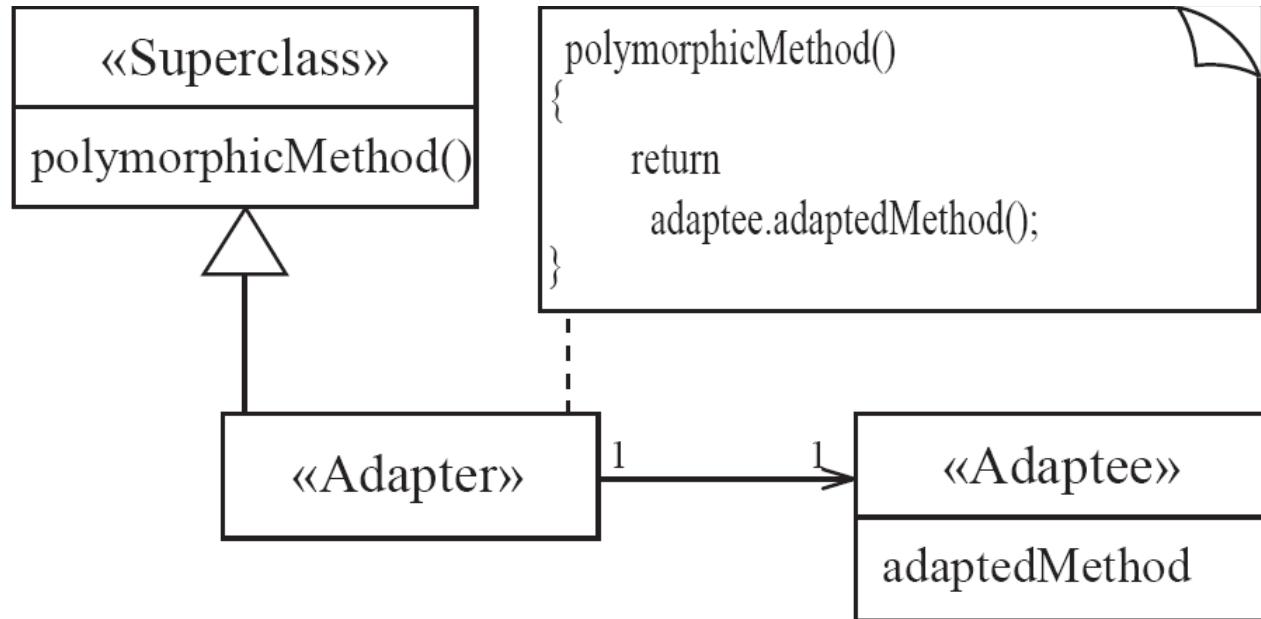
```
return specificFlight.regularFlight.flightNumber();
```

6.8 Le patron Adaptateur

- **Contexte:**
 - Vous créez une hiérarchie d'héritage et souhaitez y incorporer une classe existante (écrit par quelqu'un d'autre).
 - Cette classe réutilisée fait aussi souvent déjà partie de sa propre hiérarchie d'héritage.
- **Problème:**
 - Comment bénéficier du polymorphisme en réutilisant une classe dont
 - les méthodes ont les même fonctions
 - mais *pas* la même signature que celles de la hiérarchie existante?
- **Forces:**
 - Vous n'avez pas accès à l'héritage multiple ou vous ne voulez pas y avoir recours.

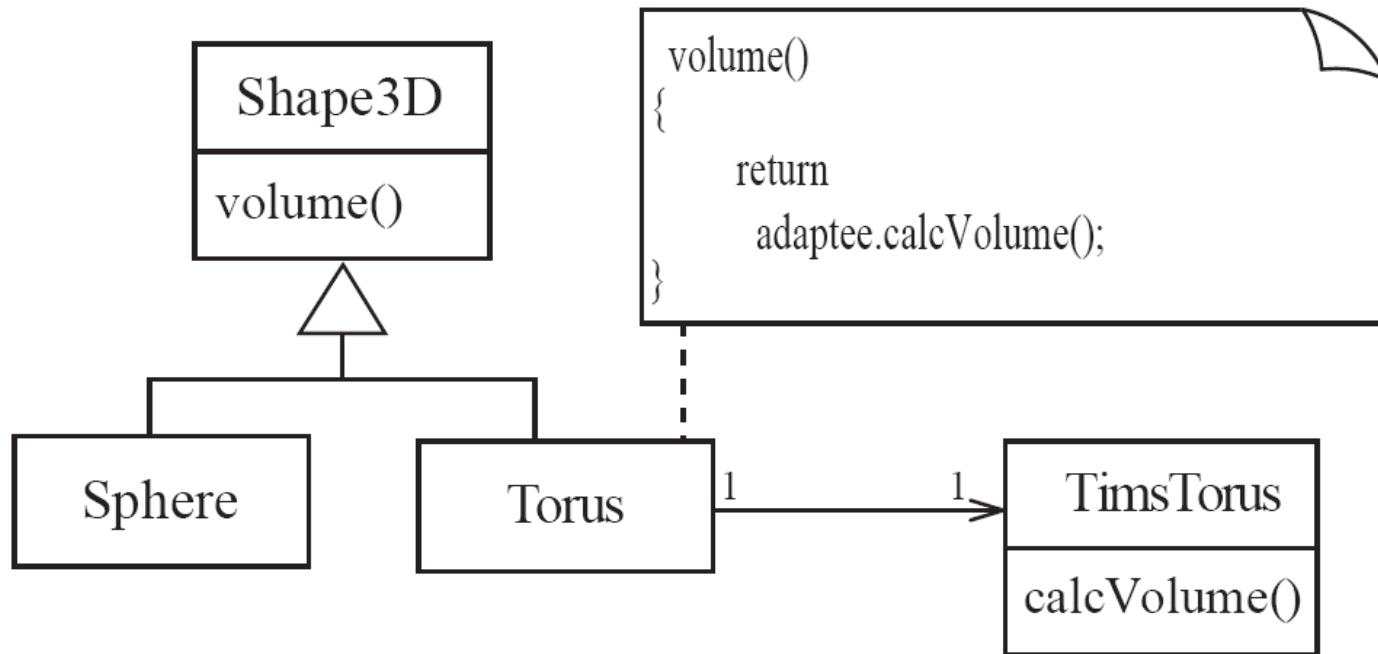
Adaptateur

- *Solution:*



Adaptateur

Exemple:

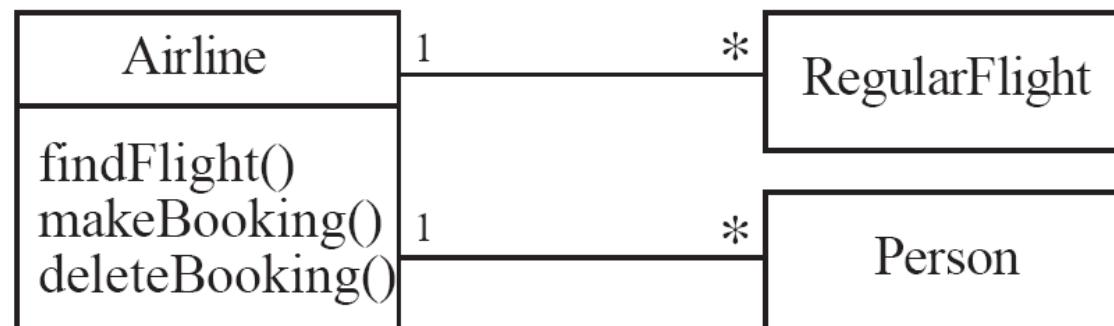
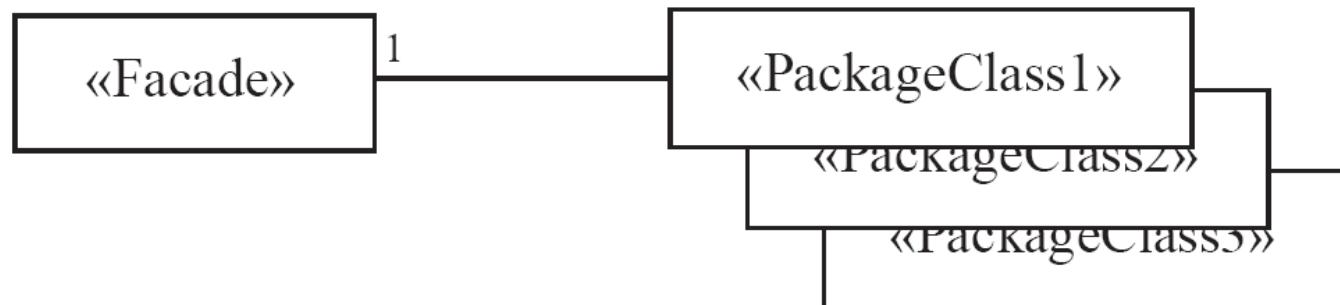


6.9 La patron Façade

- ***Contexte:***
 - Souvent, une application contient plusieurs paquetages complexes.
 - Un programmeur travaillant avec une librairie de classes doit manipuler de nombreuses classes
- ***Problème:***
 - Comment simplifier la tâche des programmeurs lorsqu'ils interagissent avec une librairie complexe?
- ***Forces:***
 - Il est difficile pour un programmeur de comprendre et d'utiliser un sous-système entier
 - Si plusieurs classes d'une application appellent les méthodes de cette librairie, alors toute modification à celle-ci nécessitera une revue complète de tout le système.

Façade

- *Solution:*



6.10 Le patron Immuable

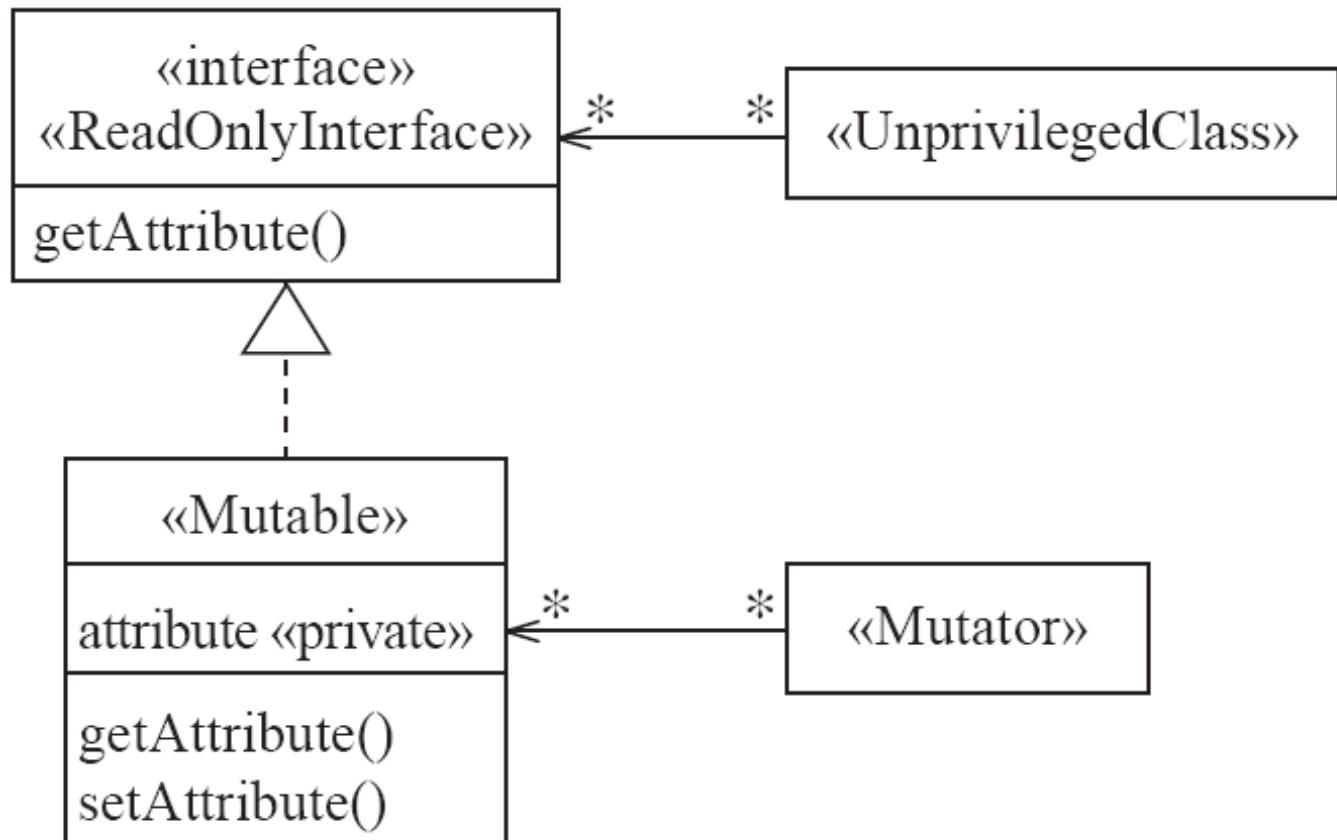
- ***Context:***
 - Un objet immuable est un objet dont l'état ne change jamais
- ***Problème:***
 - Comment créer un objet dont les instances sont immuables?
- ***Forces:***
 - Il ne doit exister aucun moyen d'altérer l'état d'un objet immuable
- ***Solution:***
 - S'assurer que le constructeur d'un objet immuable est le seul endroit où les valeurs d'un objet sont fixées.
 - Aucune méthode ne doit modifier l'état de l'objet.
 - Si une méthode devrait avoir pour effet un changement d'état, alors une nouvelle instance est retournée.

6.11 Le patron en mode lecture seule

- ***Contexte:***
 - Il faut parfois avoir certaines classes privilégiées ayant la capacité de modifier un objet immuable
- ***Problème:***
 - Comment permettre une situation où une classe est en mode lecture seule pour certaines classes tout en étant modifiable par d'autres?
- ***Forces:***
 - Restreindre l'accès par les mot-clés **public**, **protected** et **private** n'est pas adéquat.
 - Rendre **public** une méthode, la rend accessible à tous

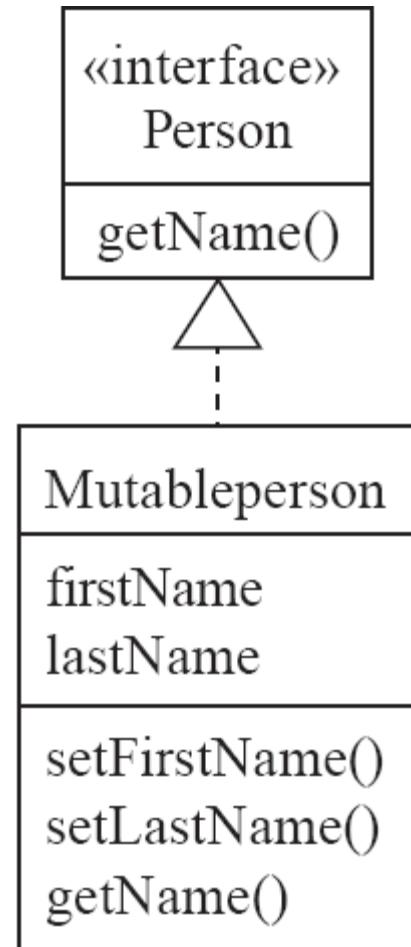
L'interface en mode lecture seule

- **Solution:**



L'interface en mode lecture seule

Example:



L'interface en mode lecture seule

Anti-patrons:

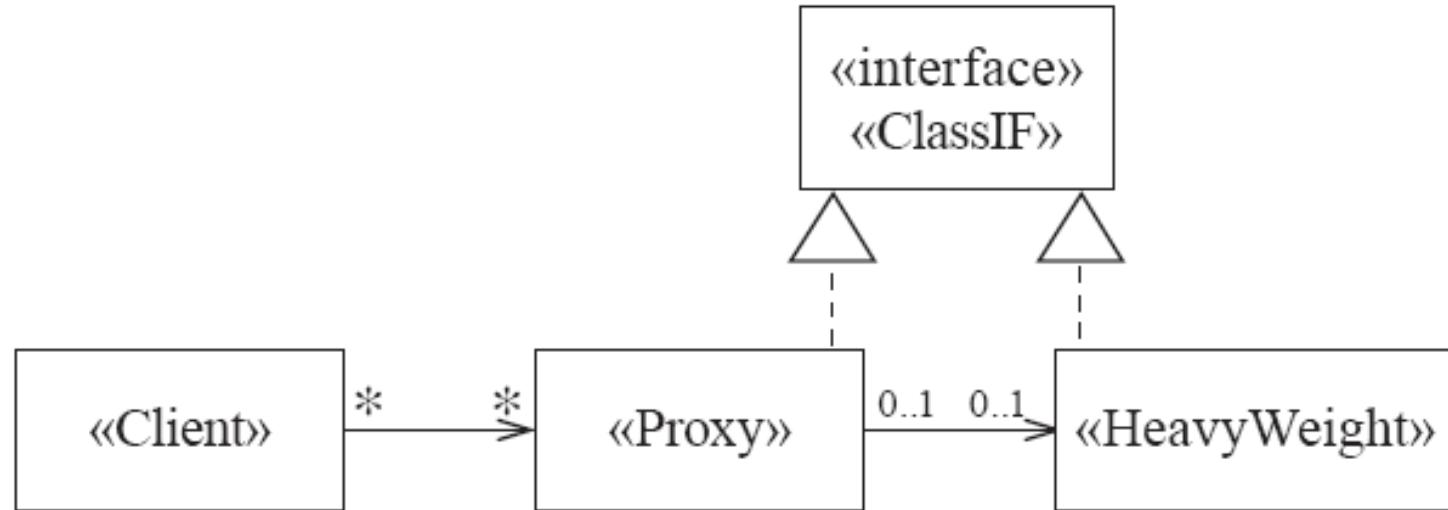
- Faire de la classe en mode lecture seule, une sous-classe de la classe immuable
- Redéfinir toutes les méthodes modifiantes de façon à ce qu'elle lancent une exception

6.12 Le patron Mandataire

- **Contexte:**
 - Fréquemment, il est coûteux et complexe de créer une instance de certaines classes (ce sont des classes *lourdes*).
 - Leur création exige du temps
- **Problème:**
 - Comment réduire la fréquence de création de classes lourdes?
- **Forces:**
 - En fonction des tâches à accomplir, tous les objets d'un système doivent demeurer disponibles lors de l'exécution du système.
 - Il est aussi important d'avoir des objets dont les valeurs persistent d'une exécution à l'autre

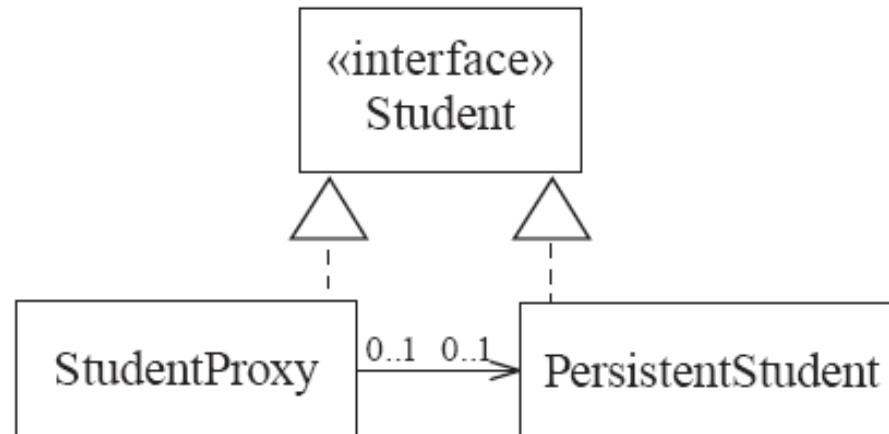
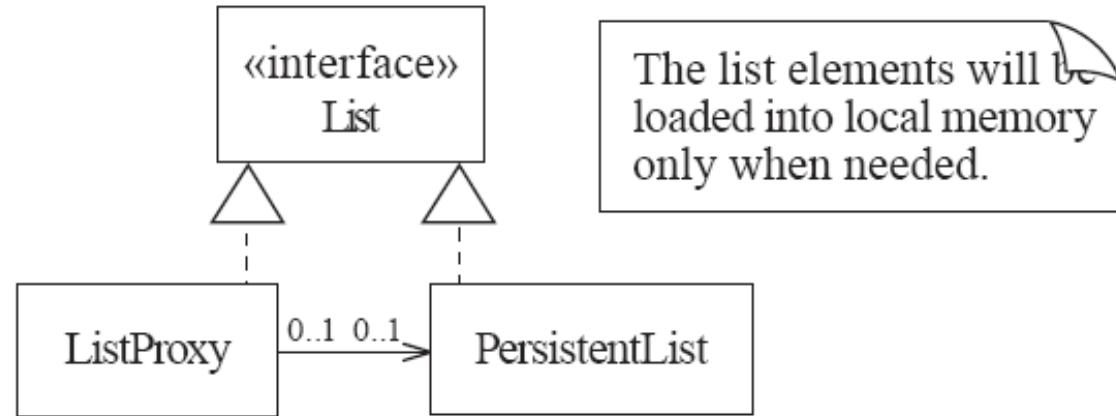
Mandataire

- *Solution:*



Mandataire

Exemples:

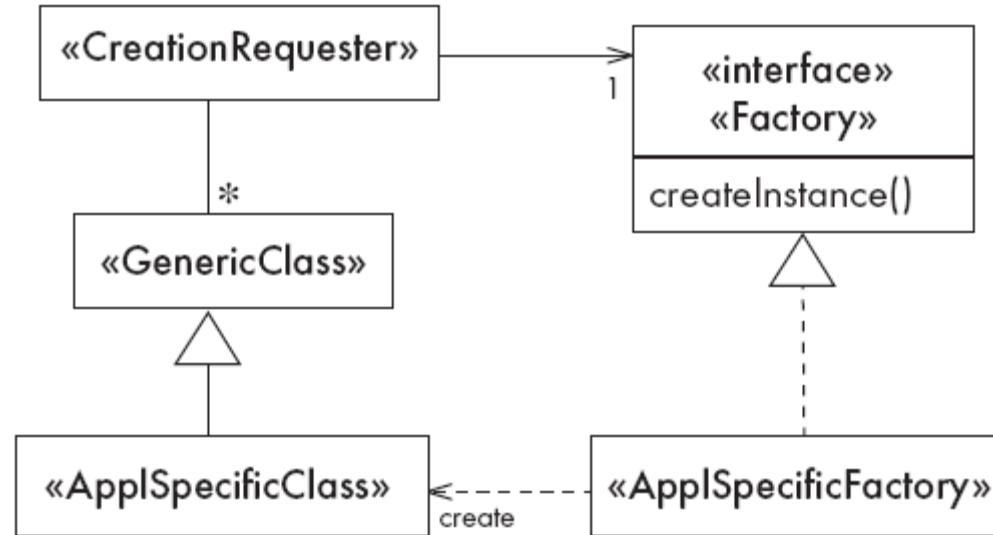


6.13 Le patron Fabrique

- **Contexte:**
 - Un cadiciel réutilisable a besoin de créer des objets; toutefois, les objets à créer dépendent de l'application.
- **Problème:**
 - Comment permettre à un programmeur d'ajouter des classes spécifiques à son application dans un système construit à partir d'un cadiciel?
- **Forces:**
 - Il faut que le cadiciel puisse créer des classes de l'application, bien qu'il ne connaît pas ces classes.
- **Solution:**
 - Le cadiciel délègue la création des classes à une classe spécialisée appelée la *Fabrique*.
 - La fabrique est une interface générique définie dans le cadiciel.
 - Cette interface contient une méthode dont le but est de créer des instances de sous-classes d'une classe générique.

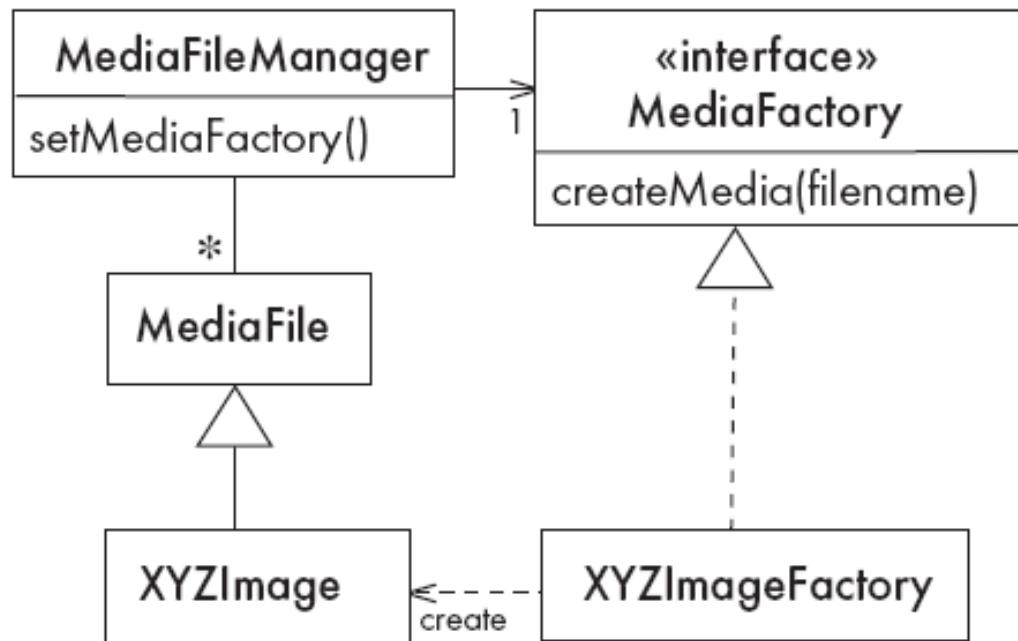
La Fabrique

Solution



La Fabrique

Exemple



6.15 Difficultés et Risques lors de la création de diagrammes de classes

- **Les patrons ne sont pas une panacé:**
 - Il ne faut pas aveuglément appliquer les patrons dès que l'occasion semble se présenter.
 - Ceci peut mener à des décisions inappropriées.
- *Résolution:*
 - *Toujours bien comprendre les forces de chacun des patrons à appliquer.*
 - *S'assurer de bien justifier chacune des décisions prises.*

Difficultés et Risques lors de la création de diagrammes de classes

- **Développer un bon patron est difficile**
 - La création d'un bon patron est une chose difficile et exigeante.
 - Un patron mal conçu présente d'intérêt
- *Résolution:*
 - Ne pas concevoir des patrons jusqu'à ce que vous soyez suffisamment expérimenté.*
 - Suivre des formations sur l'utilisation des patrons de conception.*
 - Faites réviser les patrons que vous concevez par vos pairs.*

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 8A:

Modelling Interactions and Behaviour

(Sequence Diagrams and State Machines)

8.1 Diagrammes d'interaction

Les diagrammes d'interaction sont utilisés pour modéliser les aspects dynamiques d'un système

- Ils aident à visualiser comment le système exécute ses tâches.
- Un diagramme d'interaction est souvent construit à partir d'un diagramme de classes et un cas d'utilisation
 - L'objectif est de montrer comment un ensemble d'objets peut réaliser une tâche demandée par un acteur

Les éléments se trouvant dans un diagramme d'interaction

- Des instances de classes
 - Représentés par des rectangles comme dans les diagrammes d'instances
- Acteurs
 - Représentés par des personnages-allumettes comme dans les diagramme de cas d'utilisation
- Messages
 - Représentés par des flèches horizontales

Un exemple de diagramme de séquence

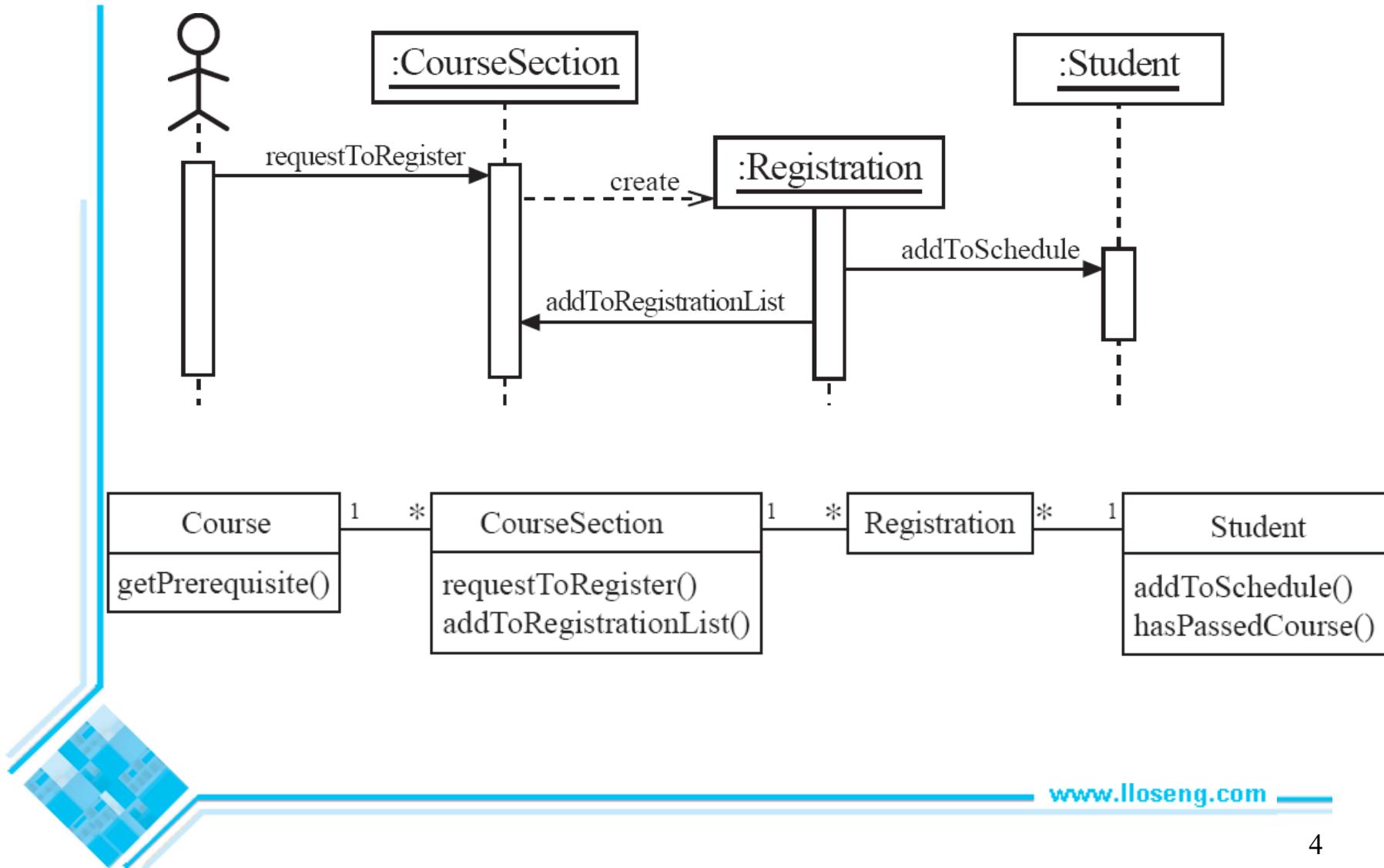


Diagramme de séquence

Un diagramme de séquence montre la séquence temporelle d'échange de message entre l'acteur et les objets réalisant une certaines tâche

- Les objets sont disposés horizontalement
- L'acteur qui initie l'interaction se trouve généralement à l'extrême gauche
- La dimension verticale représente le temps
- Une ligne verticale, appelée *ligne de vie*, est accrochée à chaque objet ou acteur
- La ligne de vie s'épaissie pour devenir une boîte d'activation lorsque l'objet est actif, i.e., durant la *période d'activation*.
- Un message est représenté par une flèche joignant deux boîtes d'activation.
 - Un message a un nom et peut aussi avoir une liste d'arguments et une valeur de retour.

Encore le même exemple, avec plus de détails

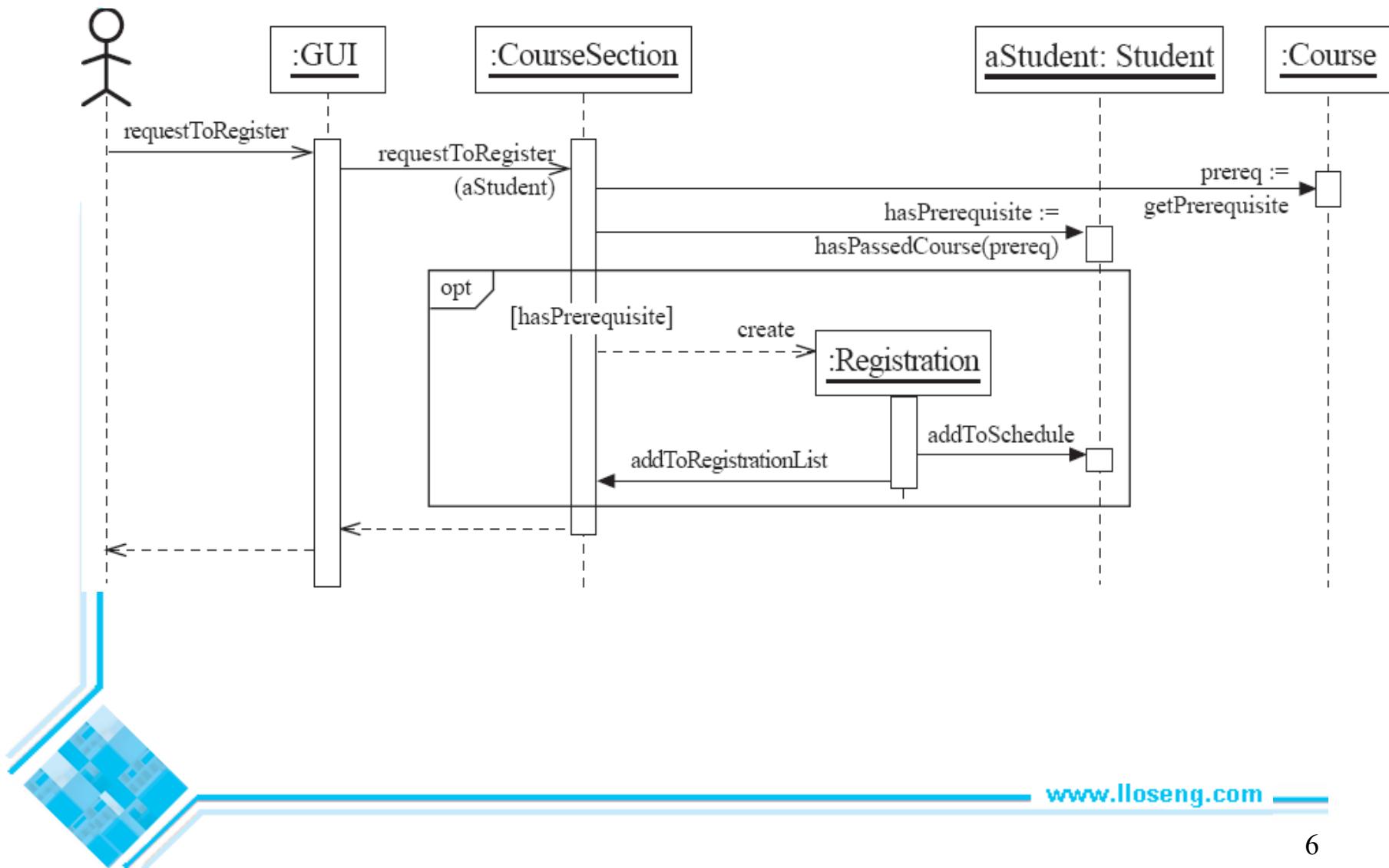


Diagramme de séquence – avec des messages répétés

- Une *itération* à travers plusieurs objets avec le fragment loop

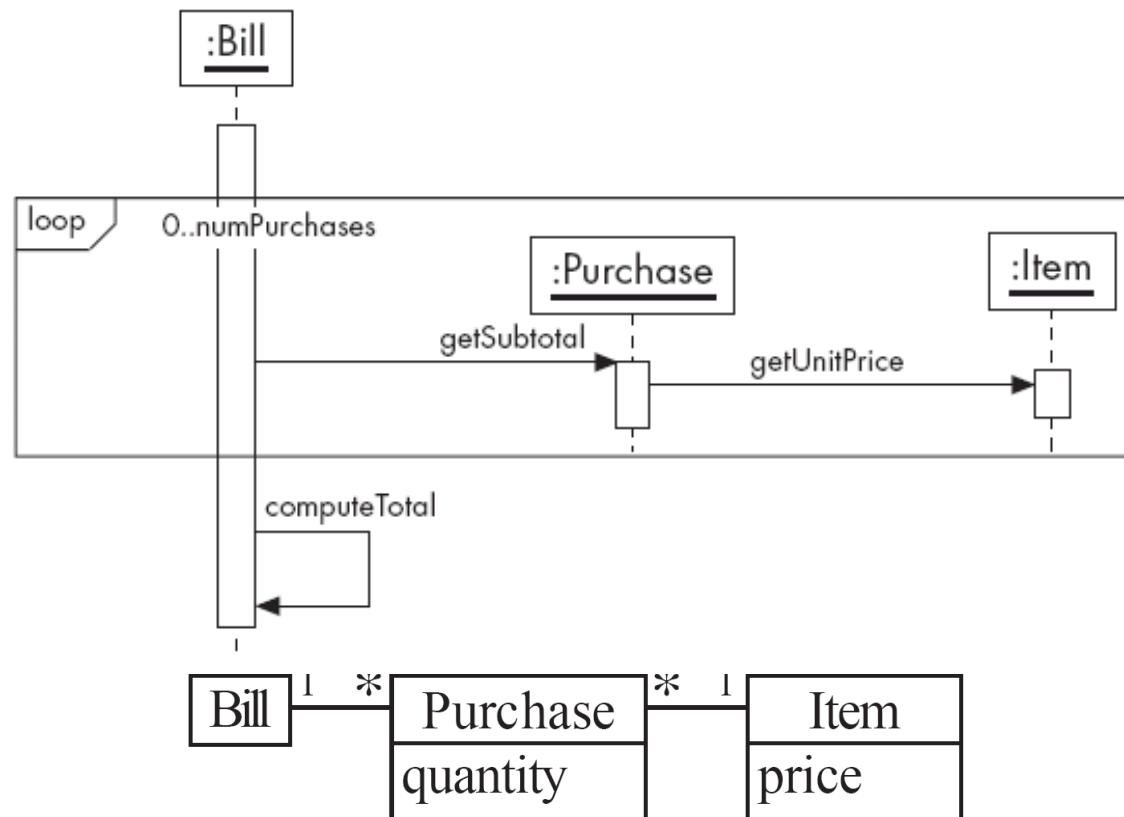
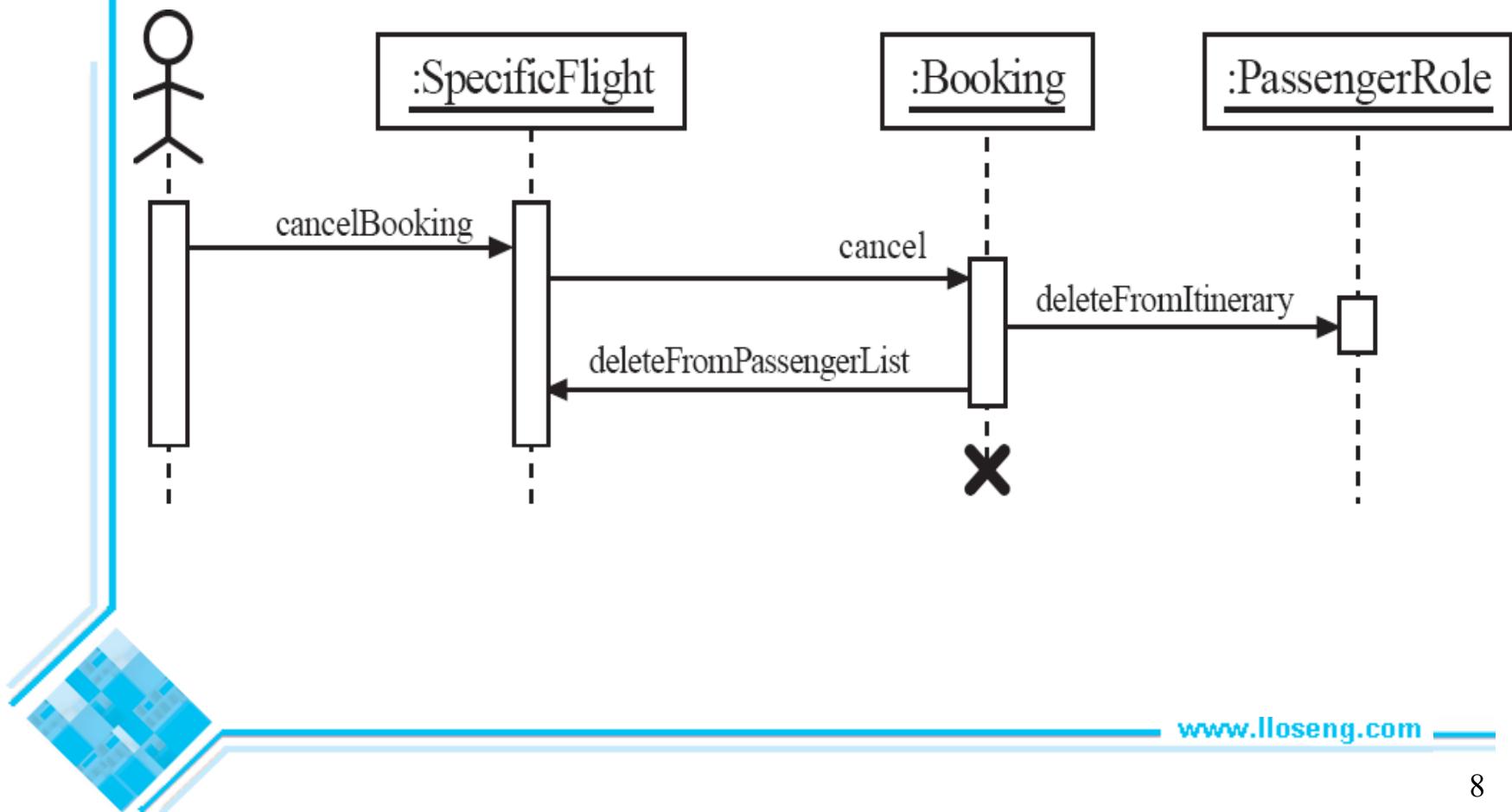


Diagramme de séquence – destruction d'un objet

- Un X à la fin d'une ligne de vie indique que l'objet a été détruit



Fragments

Les fragments permettent de décrire des diagrammes de séquence de manière compacte.

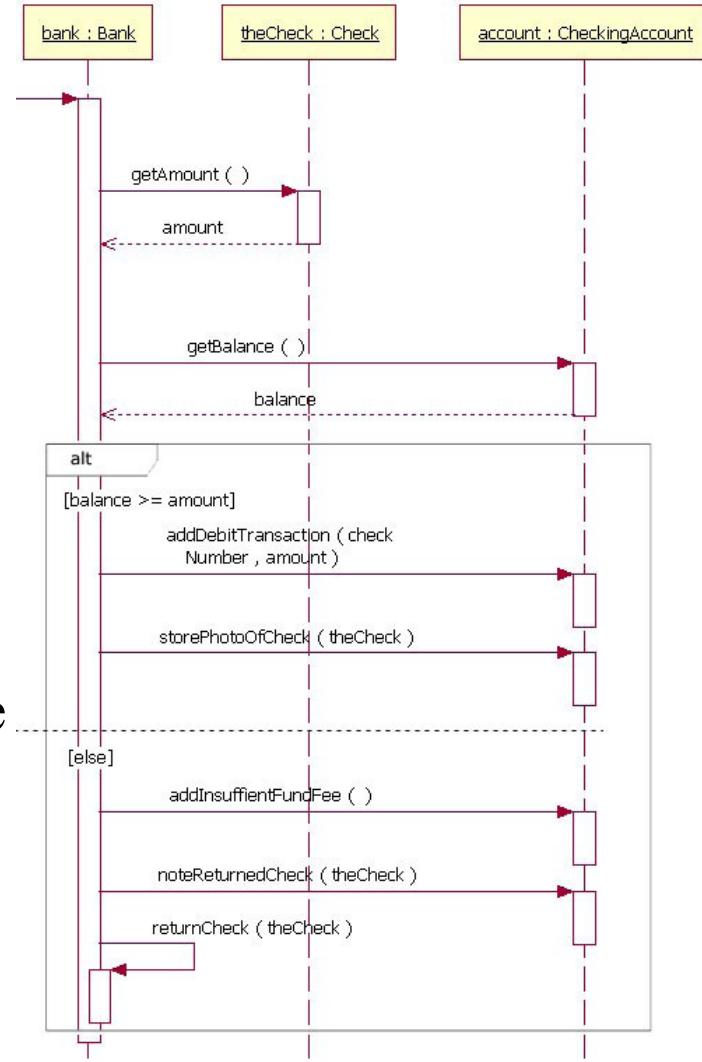
- **Alternative (alt)**, pour les alternatives avec conditions
- **Option (opt)**, pour un comportement optionnel
- **Boucle (loop)**, pour les boucles
- **Parallèle (par)**, pour un comportement concurrent

Les fragments peuvent impliquer l'ensemble des entités participant au scénario ou juste un sous-ensemble.

Fragments – Alt

Alternative (opérateur alt)

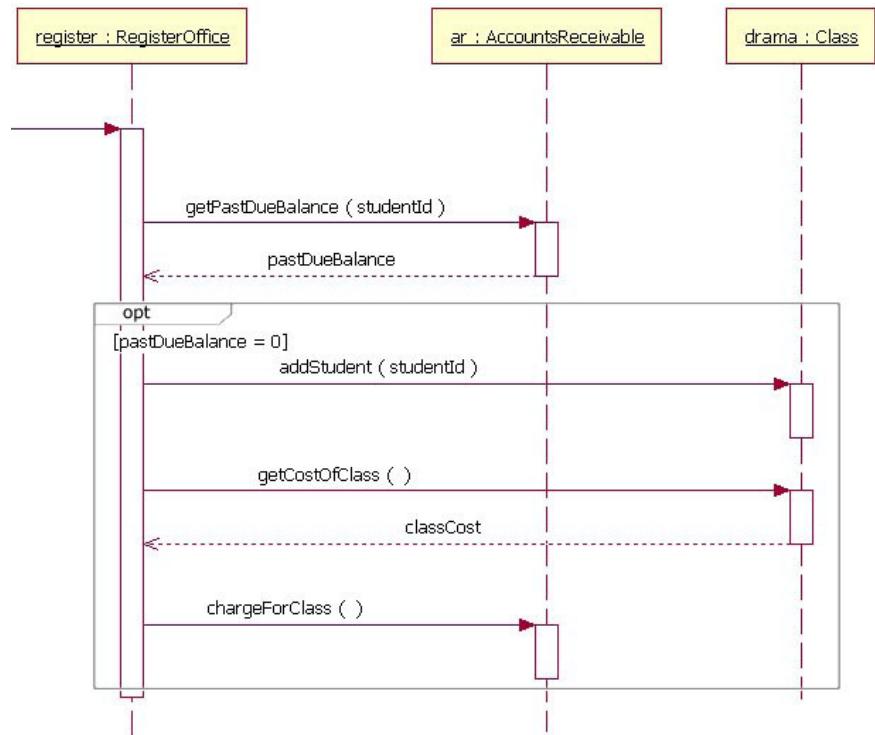
- Plusieurs opérandes (séparés par des lignes pointillées)
- Chaque opérande a une condition de garde (aucune condition implique vrai)
- Un fragment sera choisi exclusivement - de manière non déterministe si plus d'un garde évalue à vrai
- Garde spéciale: else
 - Vrai si aucune autre condition de garde n'est vraie



Fragments – Opt

Facultatif (opérateur opt)

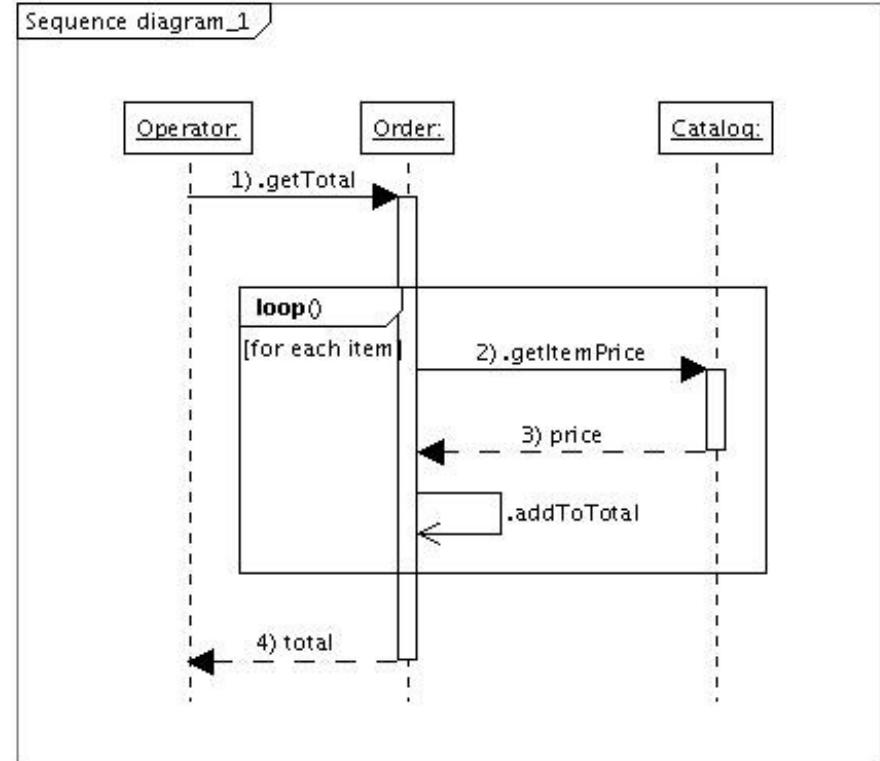
- Pour spécifier un fragment de comportement avec un garde et sans alternative
- Cas particulier d'alt
- Équivaut à un alt avec deux opérandes
 - Le premier est le même que l'opérande de l'opt
 - Le second est un opérande vide avec une garde else



Fragments – Loop

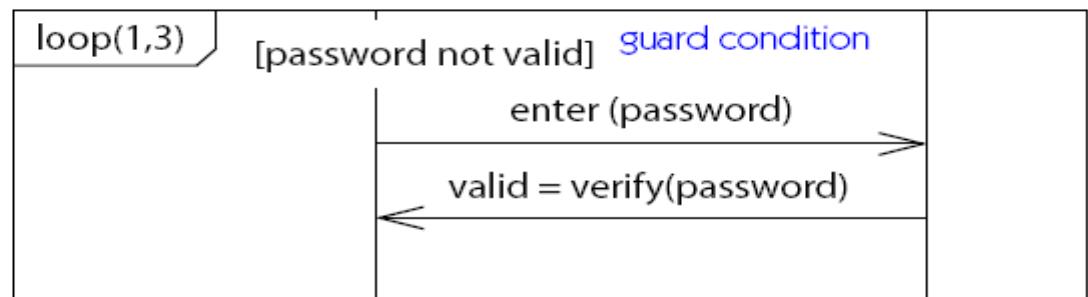
Boucle (opérateur loop)

- Le fragment de boucle peut s'exécuter plusieurs fois
- Au moins exécuté un « minimum » de fois
- Jusqu'à un nombre « maximum » tant que la condition de garde est vraie (aucune condition implique vraie)



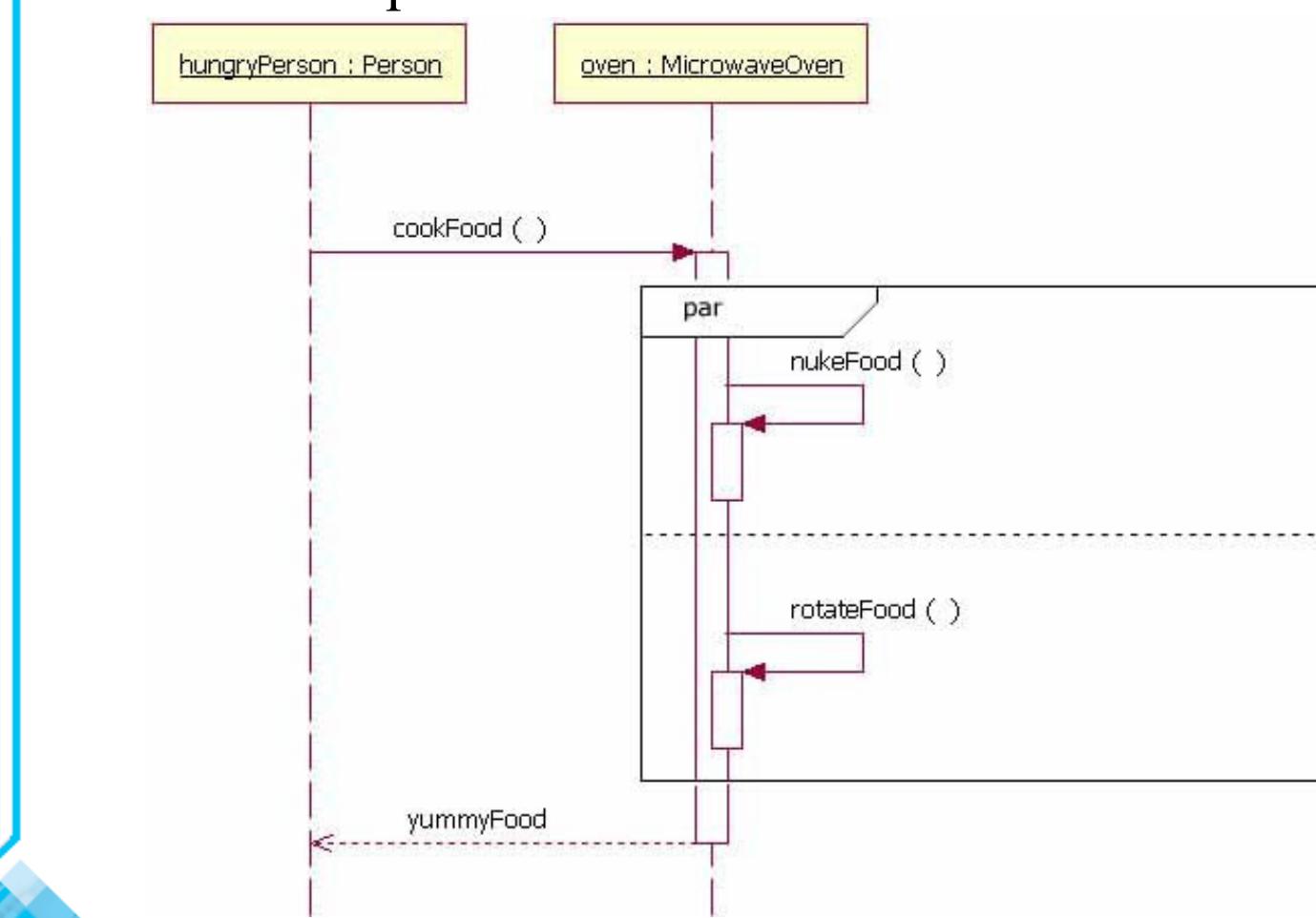
minimum, maximum count

Executes 1 to 3 times

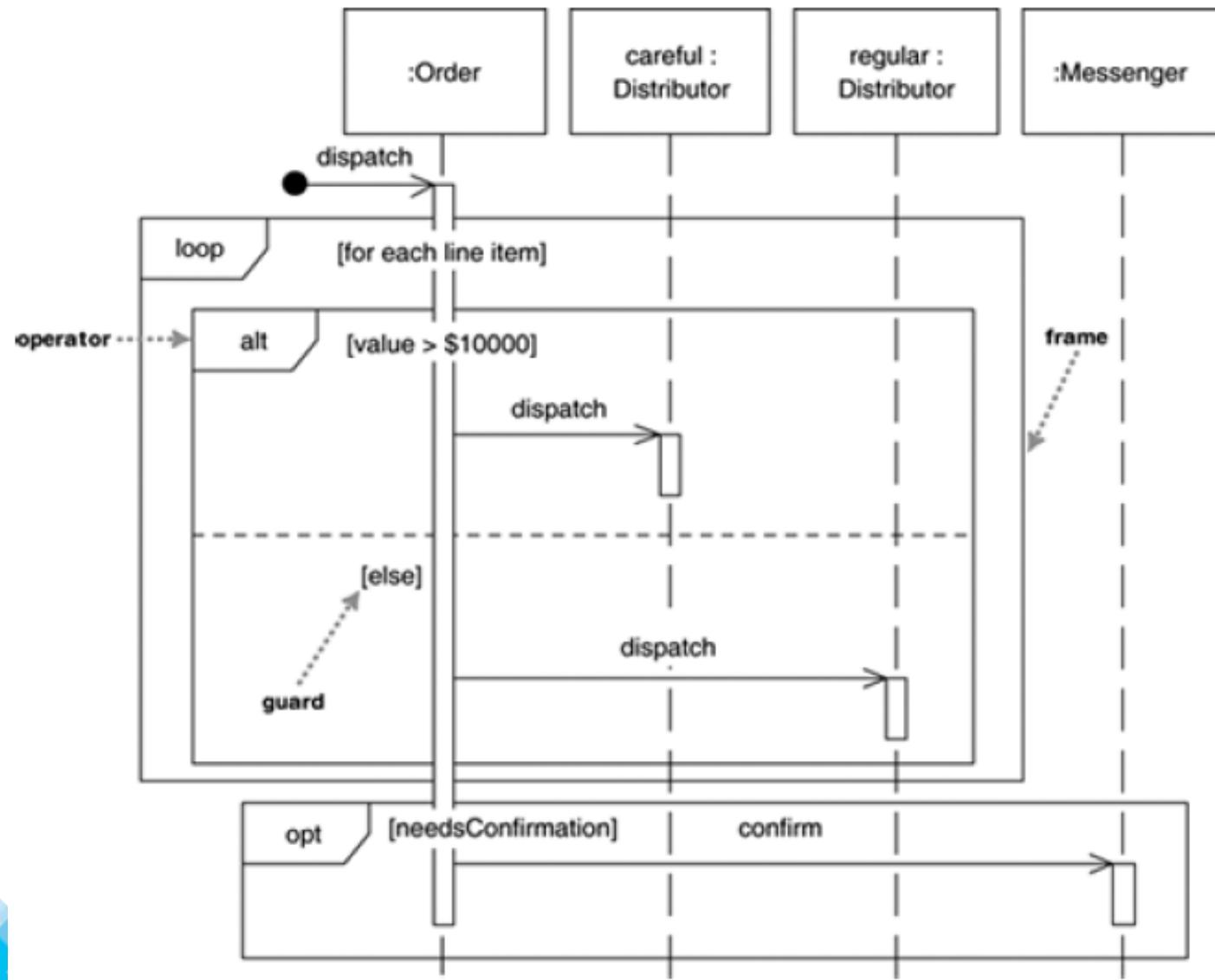


Fragments – Par

Concurrence (opérateur par): deux opérandes ou plus qui s'exécutent en parallèle



Fragments imbriqués (Nested fragments)



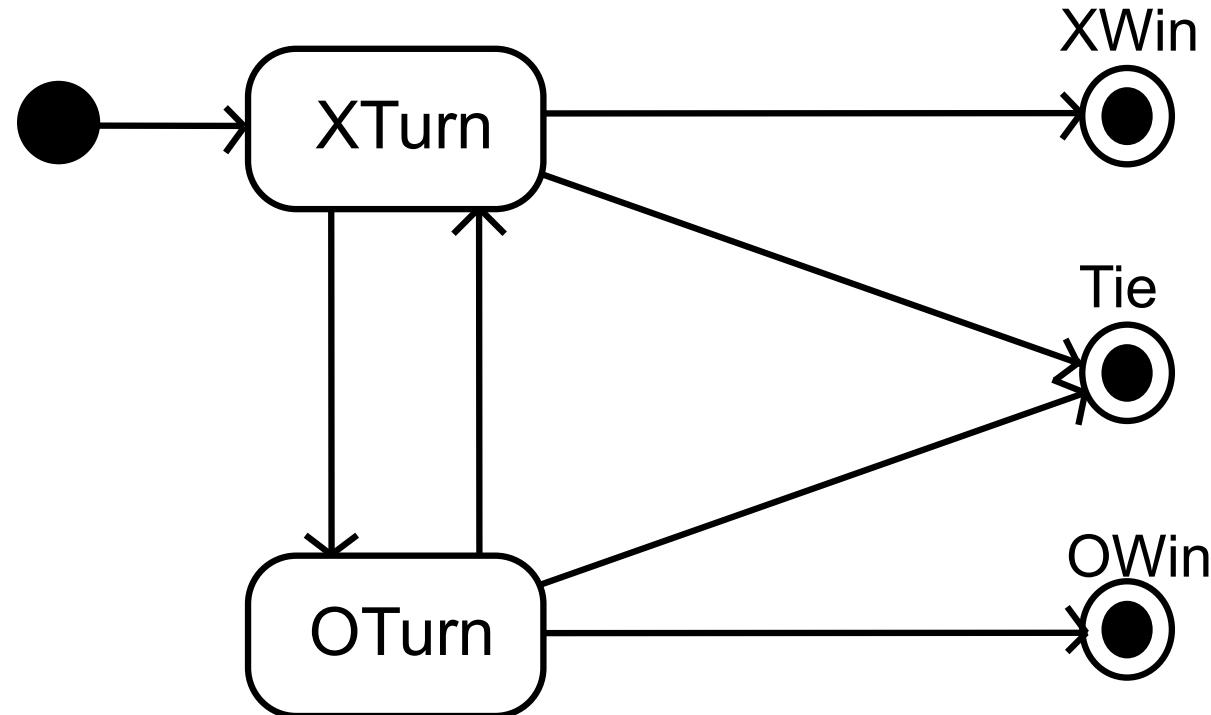
8.2 Diagrammes d'état

Un diagramme d'état décrit le comportement d'un système, d'une partie d'un système ou d'un objet.

- A tout instant, un système ou un objet se trouve dans un certain état.
 - Être dans un état donné signifie que le système se comportera d'une façon spécifique en réponse aux événements se produisant.
- Certains événements vont provoquer des changements d'états
 - Dans ce nouvel état, le système se comportera de façon différente.
- Un diagramme d'état est un graphe dans lequel les états sont des nœuds et dont les arcs représentent les transitions.

Exemple de diagramme d'état

- tic-tac-toe



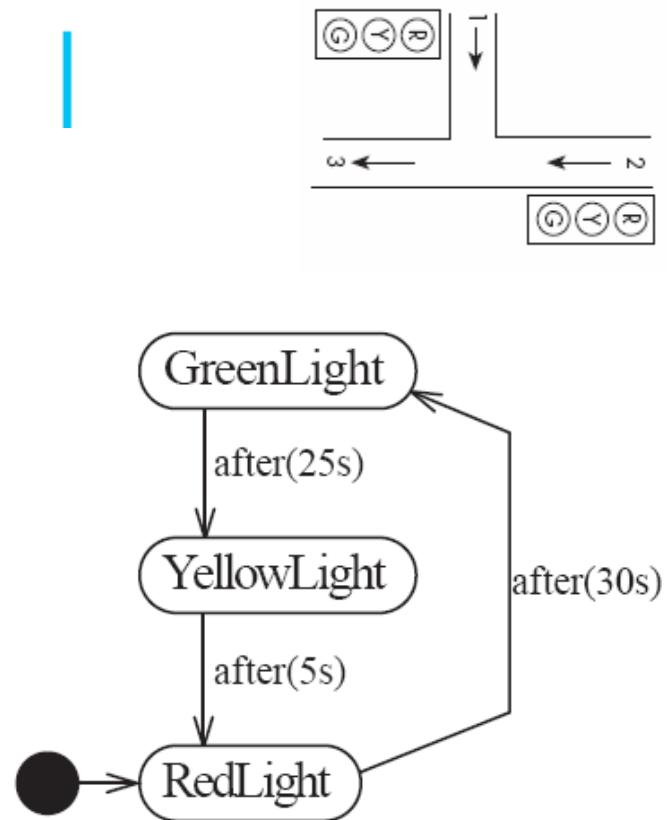
Les états

- A tout instant, le système se trouve dans un état
- Il demeura dans cet état jusqu'à l'occurrence d'un événement provoquant un changement d'état
- Un état se représente à l'aide d'un rectangle arrondi contenant le nom de cet état
- États spéciaux:
 - Un disque noir représente l'état initial
 - Un disque noir entouré d'un cercle représente un état final

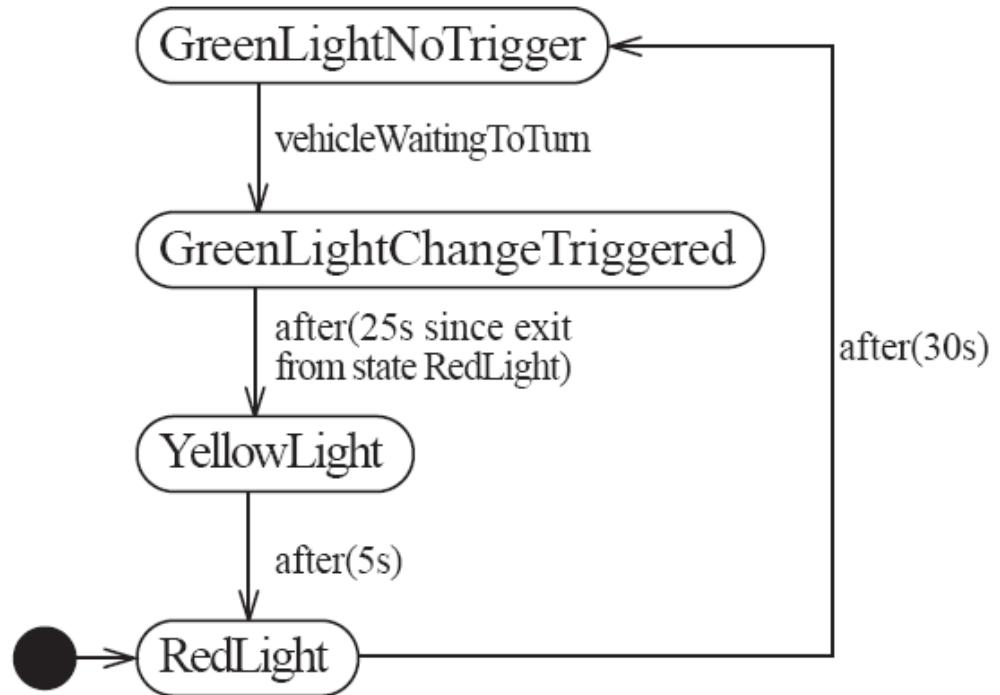
Transitions

- Une transition représente un changement d'état en réponse à un événement
 - Cette transition est considérée instantanée
- L'étiquette associée à une transition est l'événement causant ce changement d'état

Diagramme d'état – un exemple avec conditions et délais



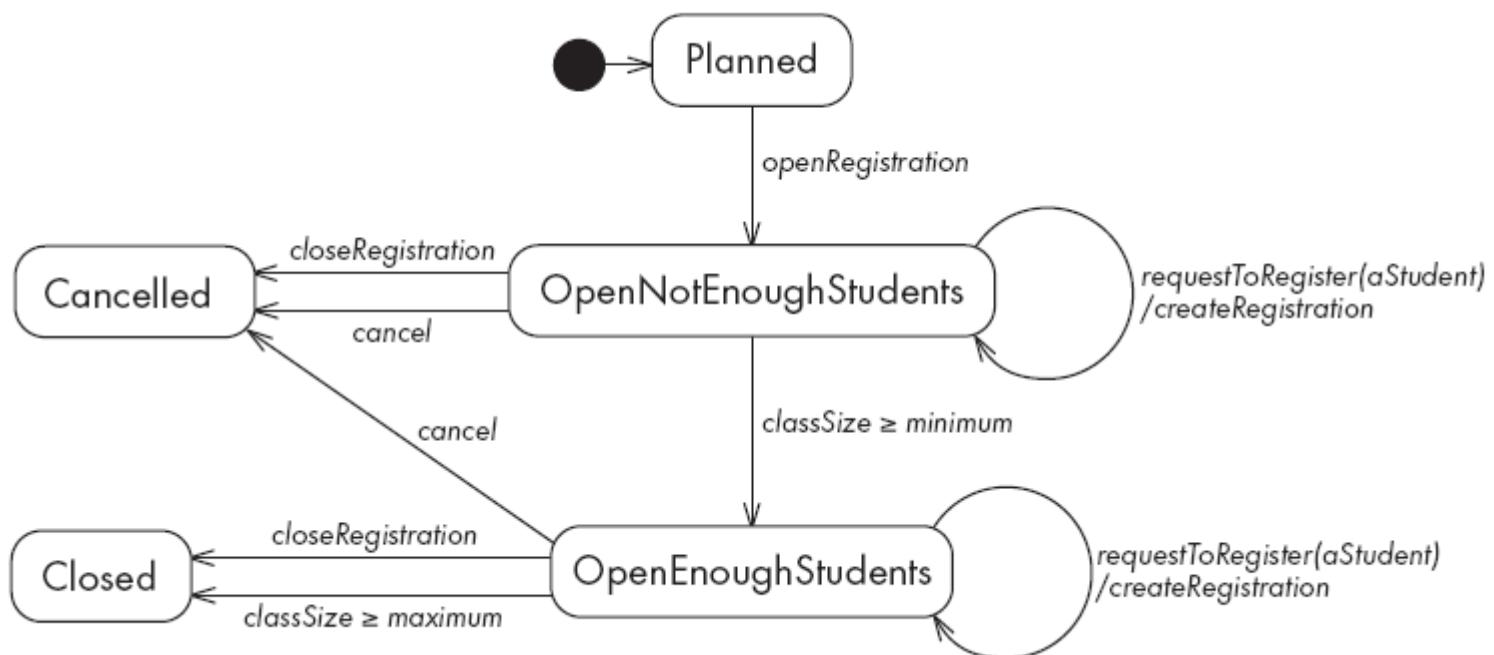
(a)



(b)



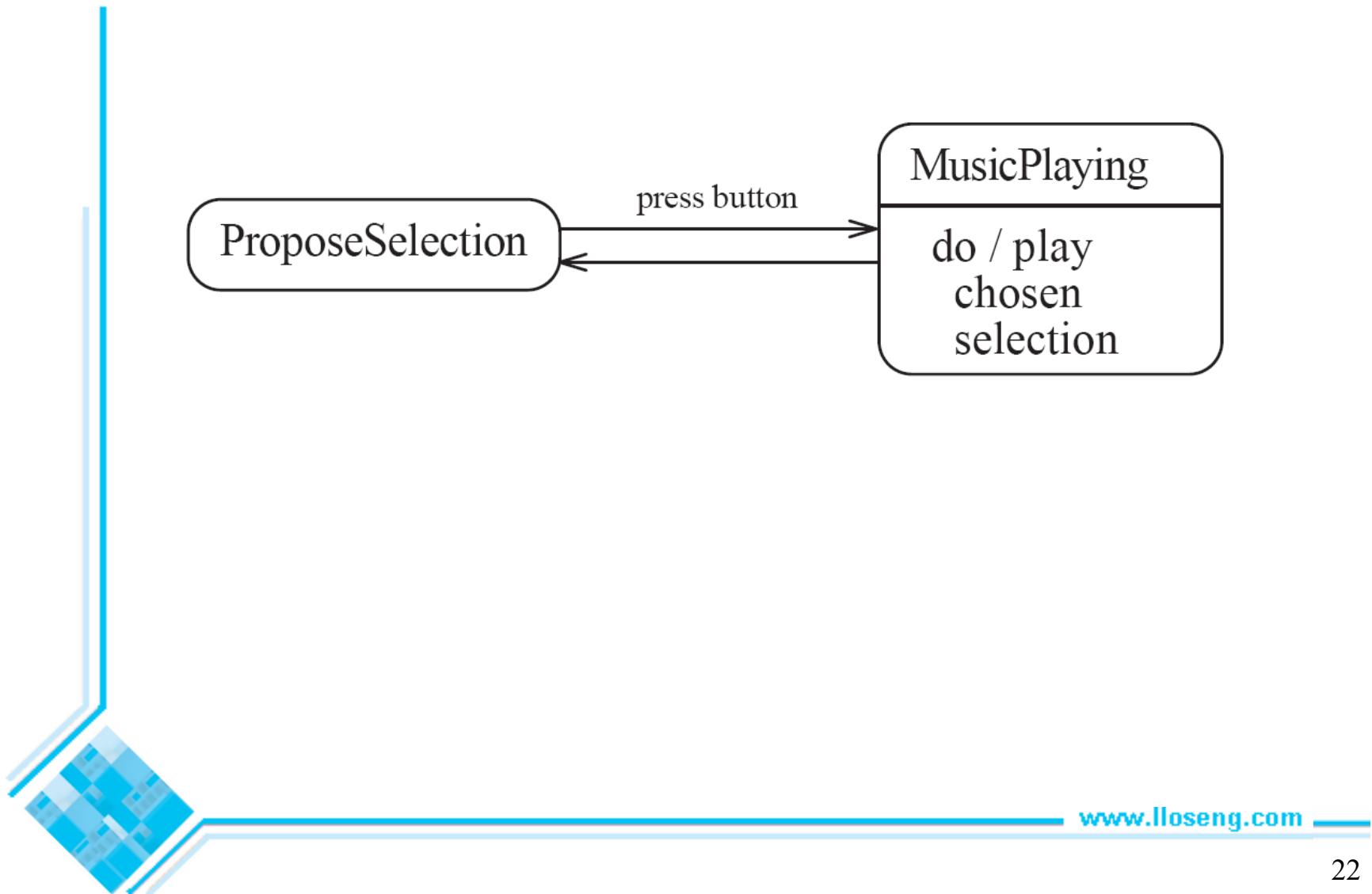
Diagramme d'état – un exemple avec transition conditionnelle



Les activités dans les diagramme d'état

- Une *activité* peut être lancée lorsque le système se trouve dans un état
 - Une activité a une durée
 - En réponse à la terminaison de l'activité, le système peut effectuer un changement d'état
 - Les transitions peuvent aussi se produire lorsque l'activité est en cours:
 - Le système doit terminer l'activité lorsqu'il quitte l'état

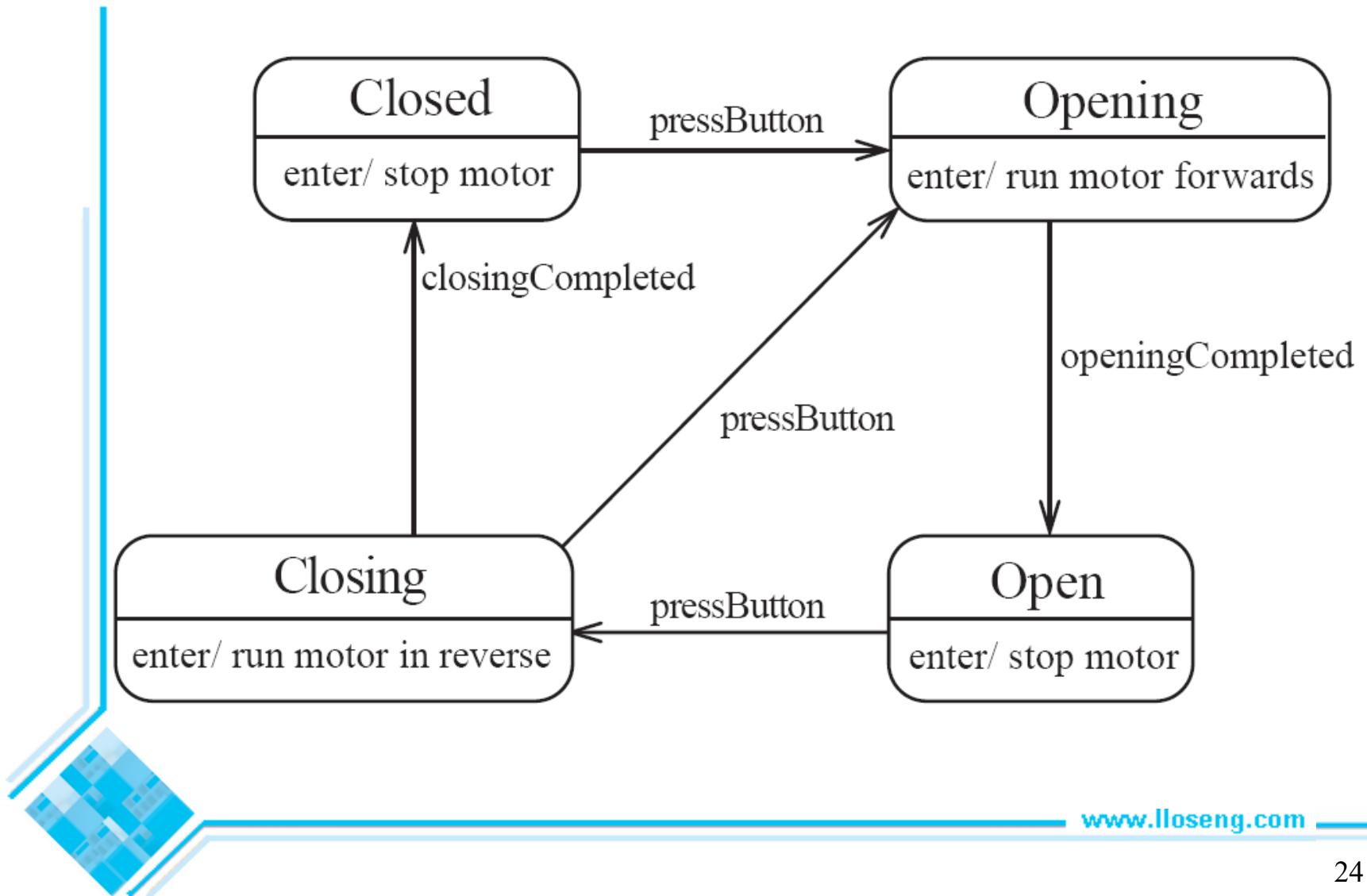
Diagramme d'état – un exemple avec activité



Les actions dans un diagramme d'état

- Une *action* se réalise de façon instantanée
 - Lorsqu'une transition se produit
 - Lorsque le système entre dans un certain état
 - Lorsque le système sort d'un certain état

Diagramme d'état – un exemple avec actions



Sous-états et diagrammes imbriqués

Un diagramme d'état peut être imbriqué dans un autre.

- Les états dans un diagramme interne sont des *sous-états*

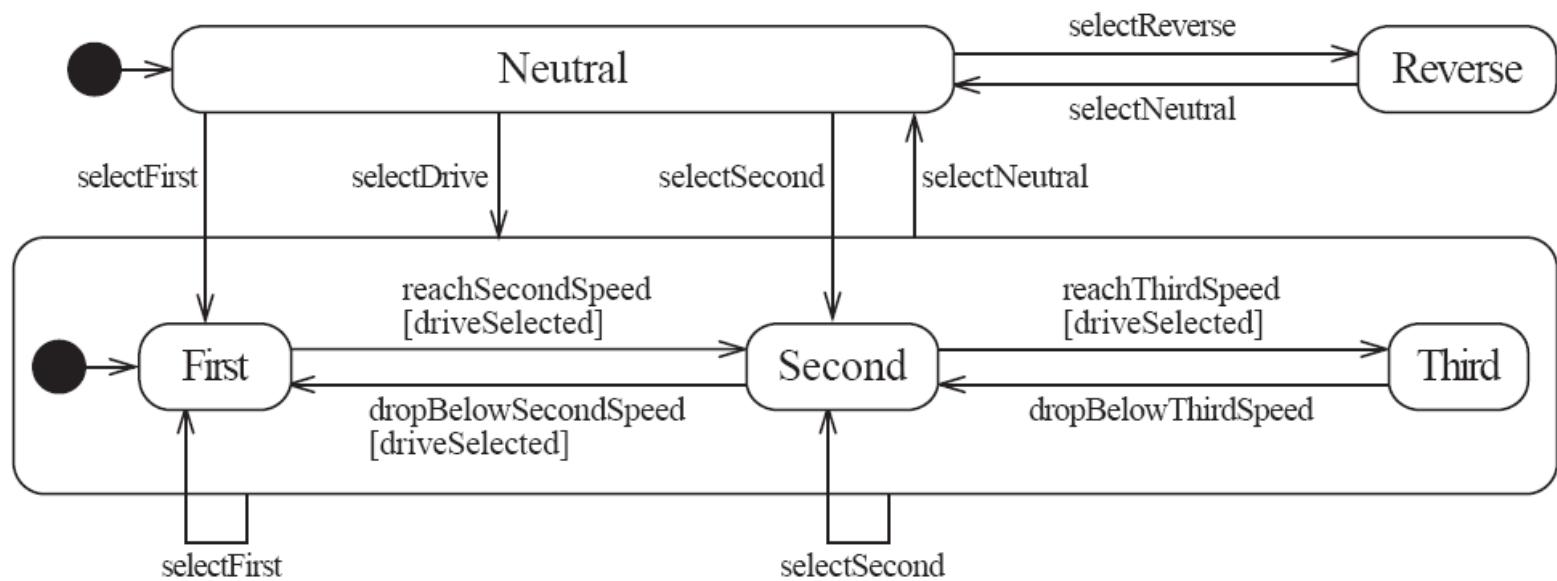
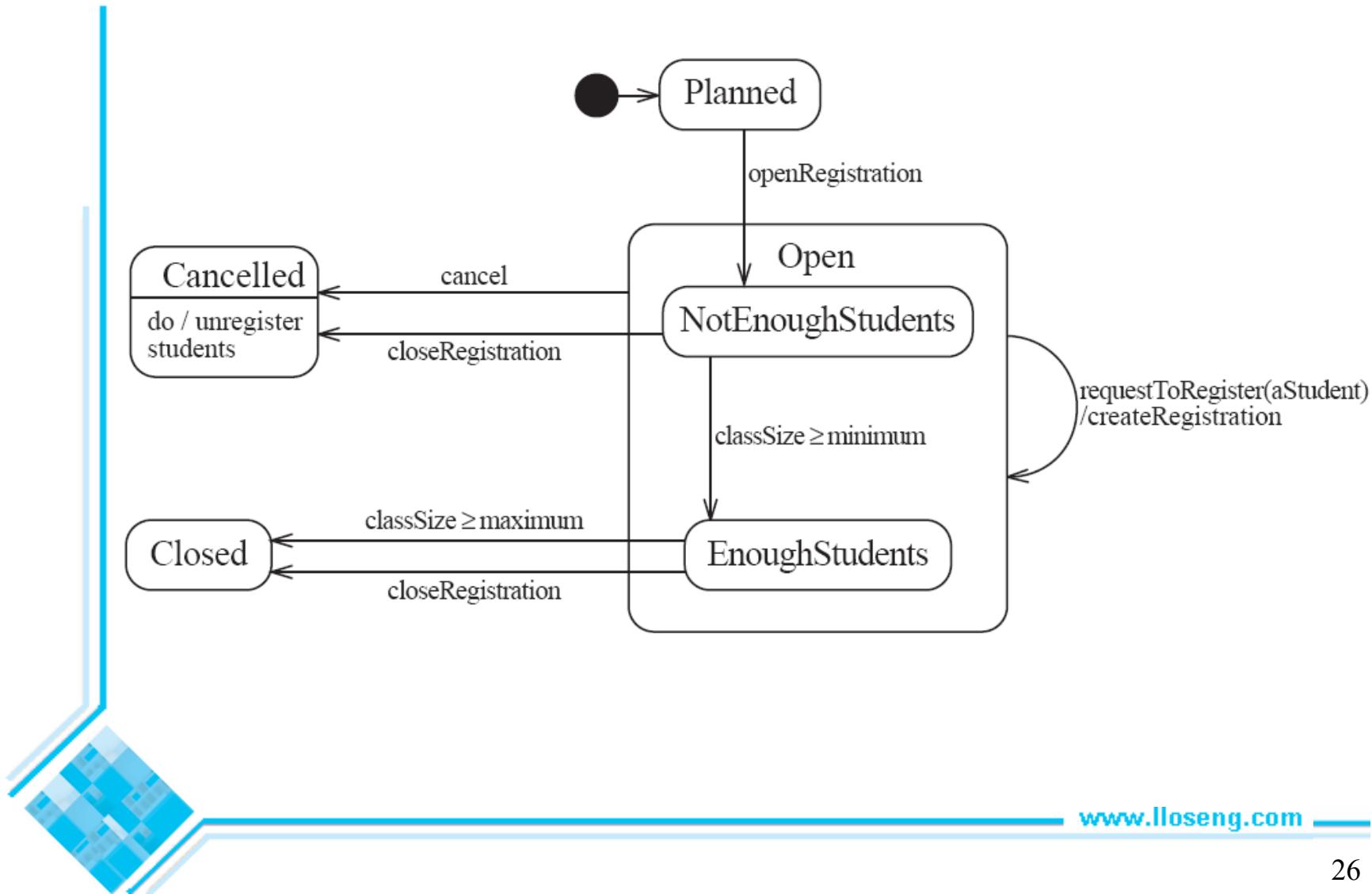
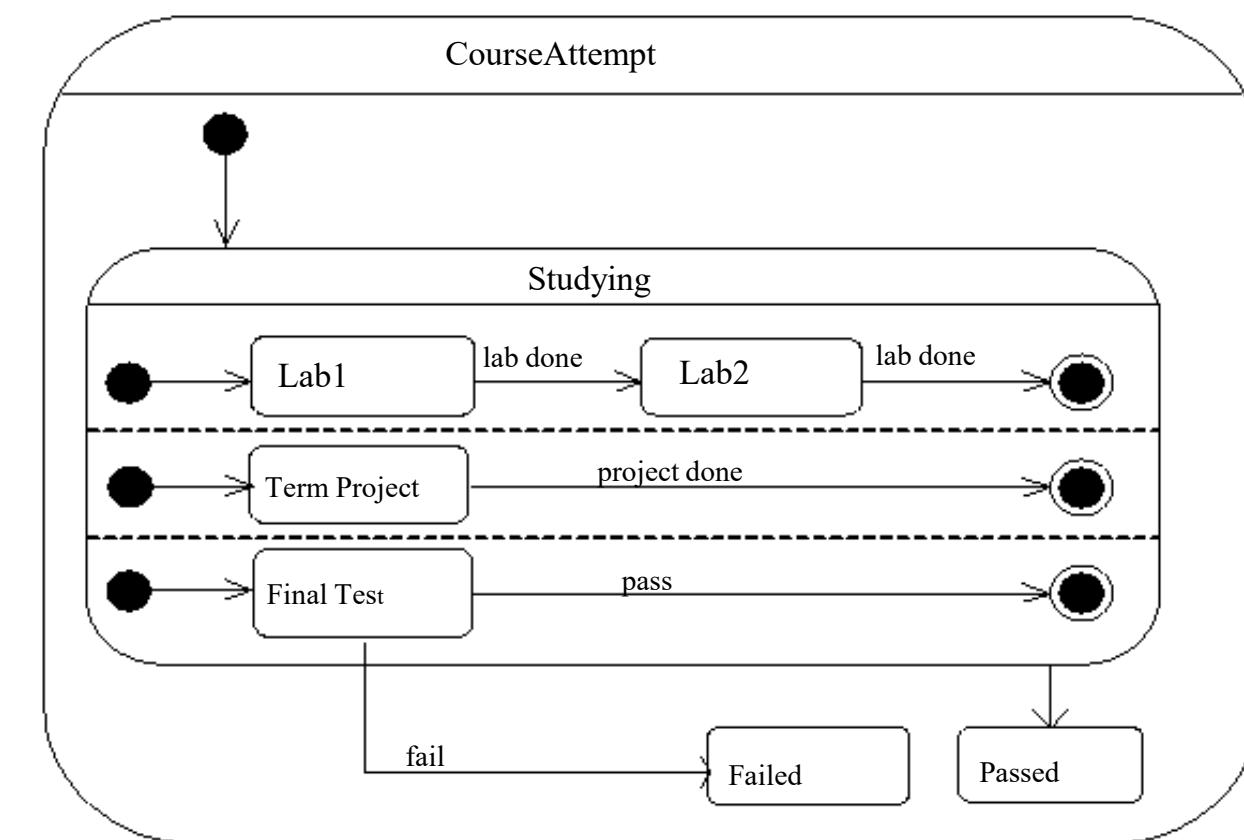


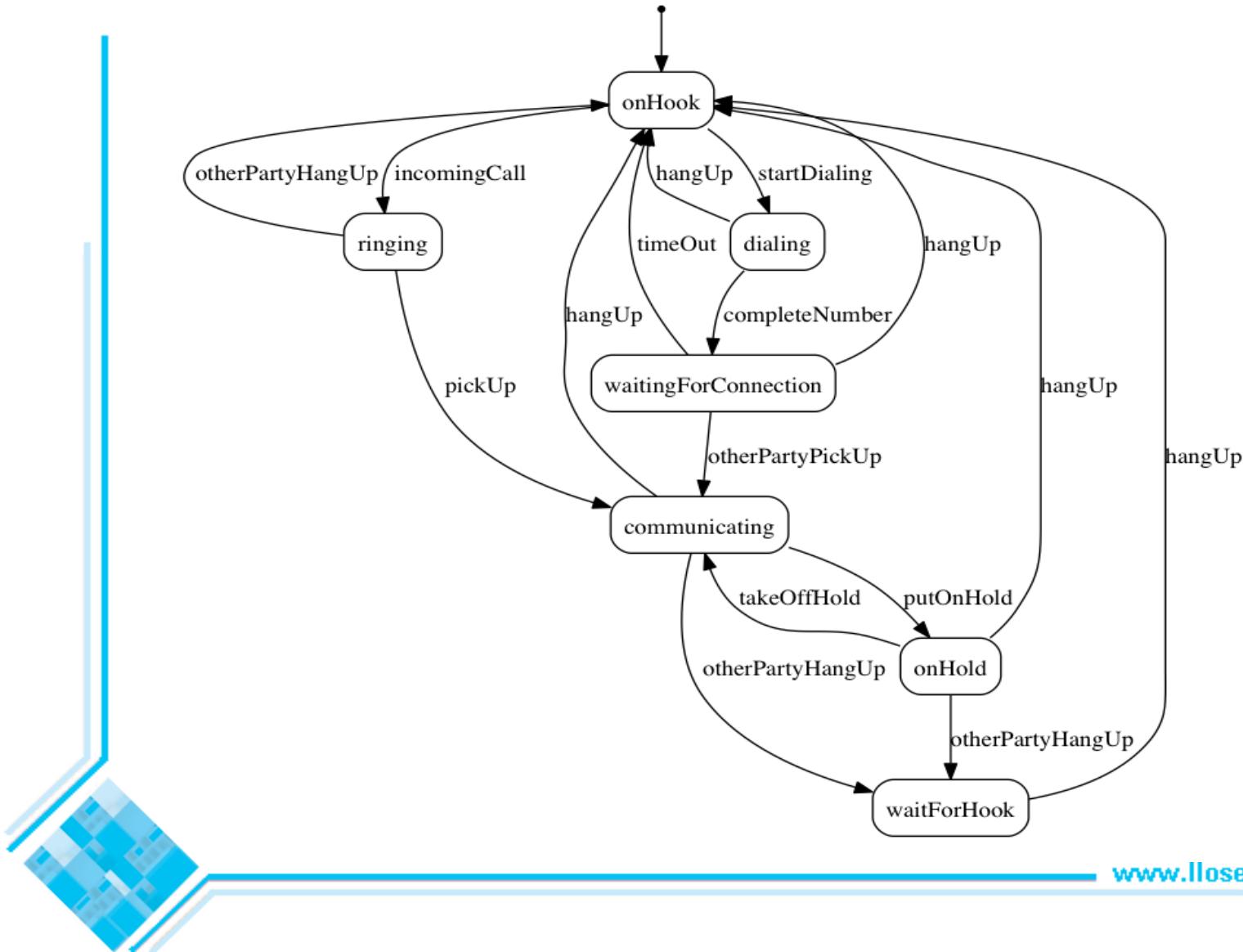
Diagramme d'état – un exemple avec sous-états



Exercice: Décrivez ce comportement



Machine d'état pour une ligne téléphonique



Code Umple pour l'exemple du ligne téléphonique

```
class phone {  
    state {  
        onHook {  
            startDialing -> dialling;  
            incomingCall -> ringing;  
        }  
        ringing {  
            pickUp -> communicating;  
            otherPartyHangUp -> onHook;  
        }  
        communicating {  
            hangUp -> onHook;  
            otherPartyHangUp -> waitForHook;  
            putOnHold -> onHold;  
        }  
        onHold {  
            hangUp -> onHook;  
            otherPartyHangUp -> waitForHook;  
            takeOffHold -> communicating;  
        }  
        dialing {  
            completeNumber ->  
            waitingForConnection;  
            hangUp -> onHook;  
        }  
        waitingForConnection {  
            otherPartyPickUp -> communicating;  
            hangUp -> onHook;  
            timeOut -> onHook;  
        }  
        waitForHook {  
            hangUp -> onHook;  
        }  
    }  
}
```

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 8B:

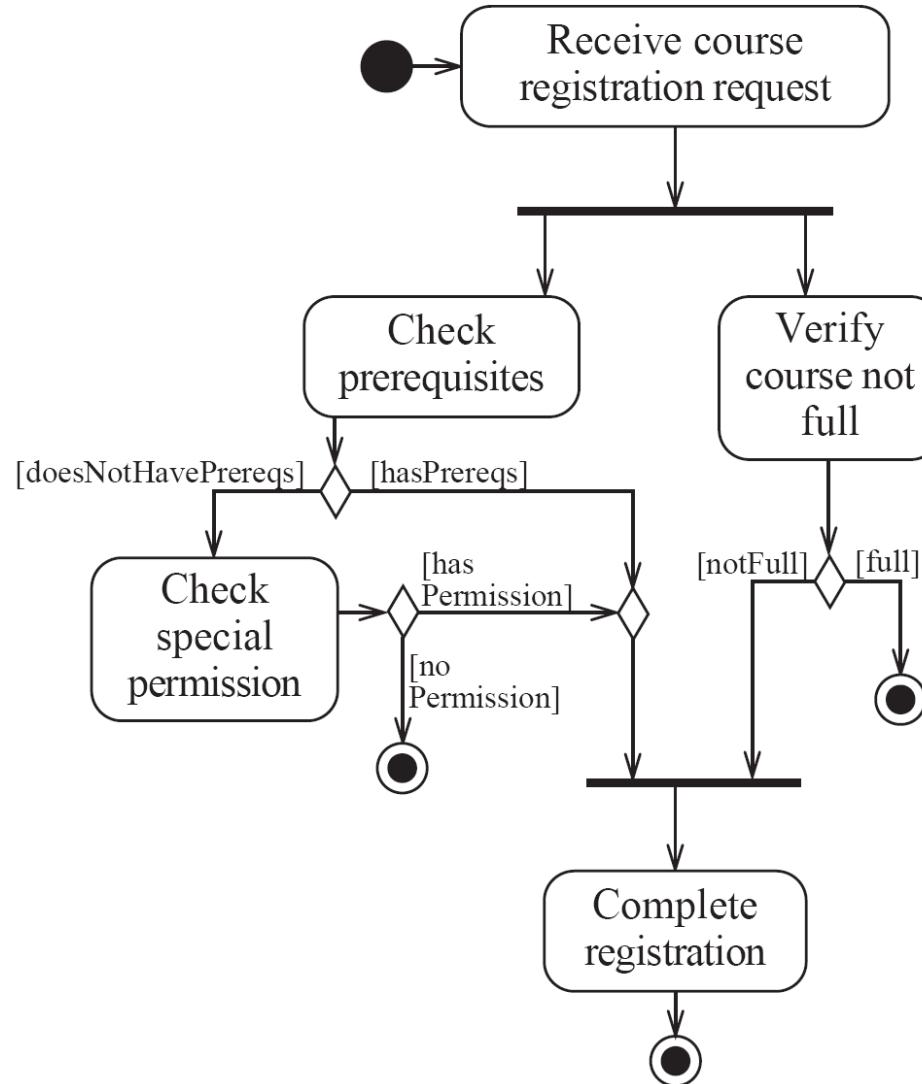
Modelling Interactions and Behaviour

(Activity Diagrams)

8.3 Diagrammes d'activité

- Un diagramme d'activité est similaire à un diagramme d'état.
 - Les transitions sont causées par la terminaison d'une activité.
- Un diagramme d'activité
 - peut servir à mieux comprendre la succession des étapes qu'un système doit accomplir afin de réaliser une certaine tâche
 - peut aussi servir à mieux comprendre les cas d'utilisations et la façon dont ils sont interreliés
 - est le plus souvent associé à plusieurs classes
- L'une des forces des diagrammes d'activité est de représenter des activités concurrentes.

Un exemple de diagramme d'activité



Représentation de la concurrence

- La concurrence dans un diagramme d'activité se représente à l'aide de points de rencontre, de fourchettes et de rendez-vous.
 - Une fourchette (*fork*) a une transition entrante et plusieurs transitions sortantes
 - L'exécution se sépare alors en différents fils d'exécution
 - Un *rendez-vous* a plusieurs transitions entrantes et sortantes
 - Lorsque toutes les transitions entrante ont été effectué alors seulement peuvent être lancée les transitions sortantes

Représentation de la concurrence

—Un point de rencontre (*join*) a de multiples transitions entrantes et une transition sortante

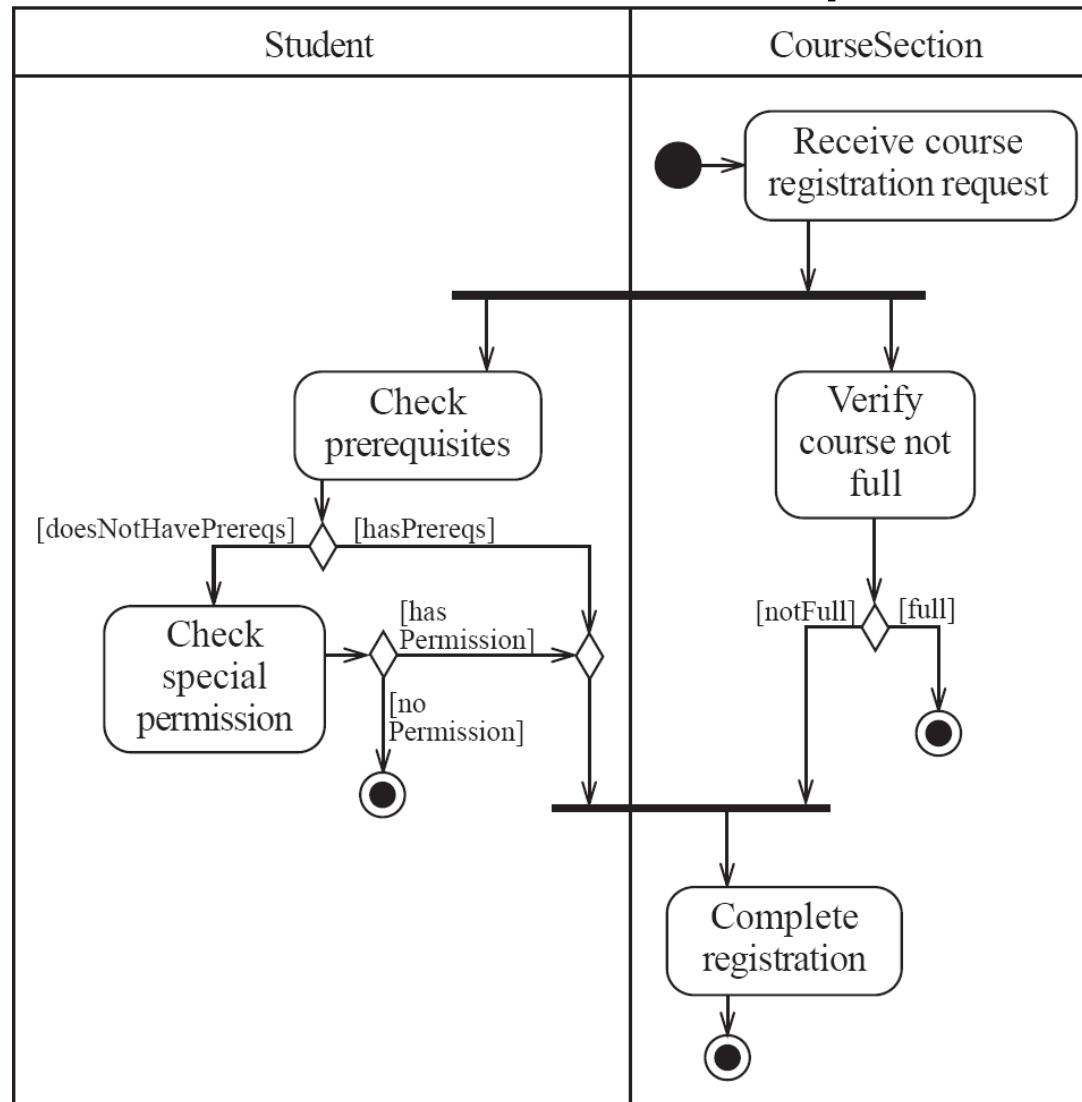
- La transition sortante s'effectue lorsque toutes les transitions entrantes se sont produites
- Chacune des transitions entrantes s'effectue dans un fil d'exécution distinct.
- Lorsque qu'une transition entrante se produit, le fil correspondant est bloqué jusqu'à ce que les autres transitions soient complétées.

Allées

Un diagramme d'activité est souvent associé à plusieurs classes

- Le partitionnement des activités à travers les différentes classes peut être explicitement montré à l'aide d'allées (*swimlanes*)

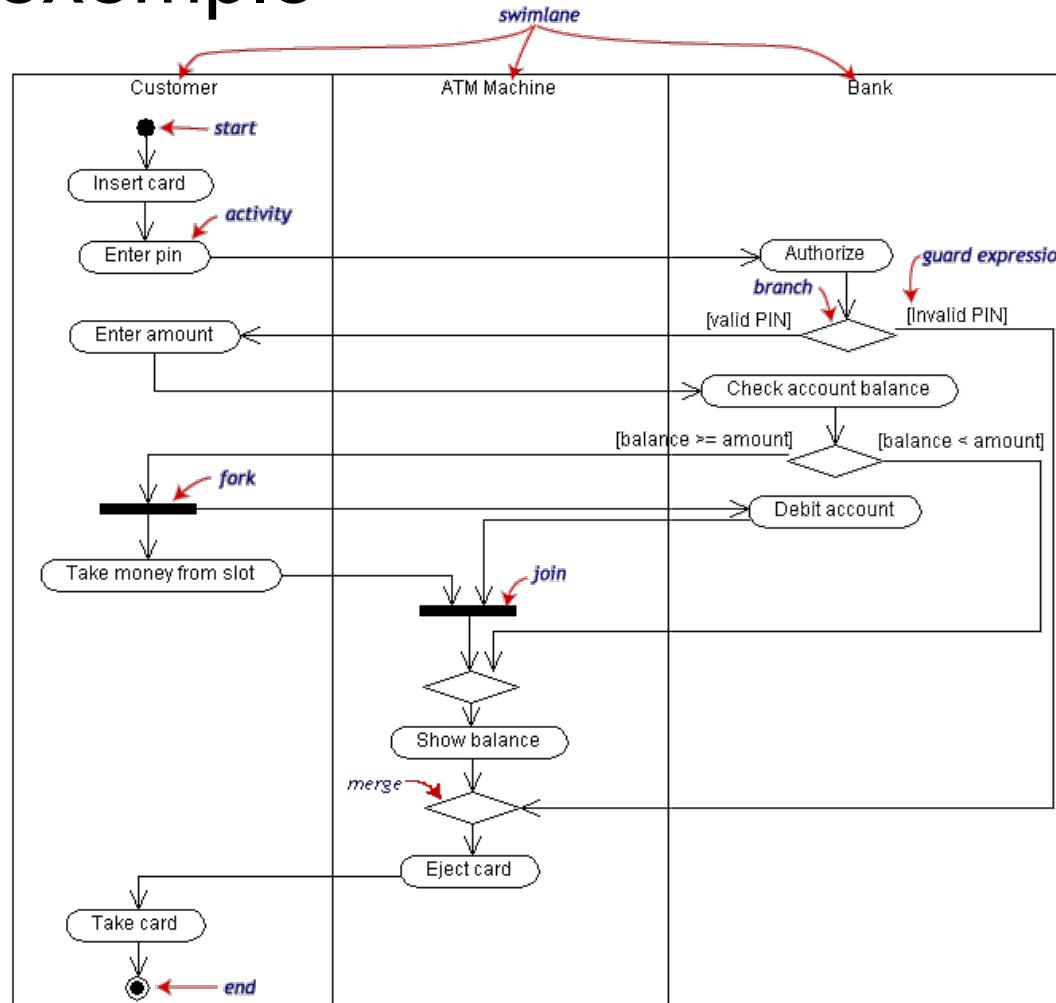
Diagramme d'activité – un exemple avec allées



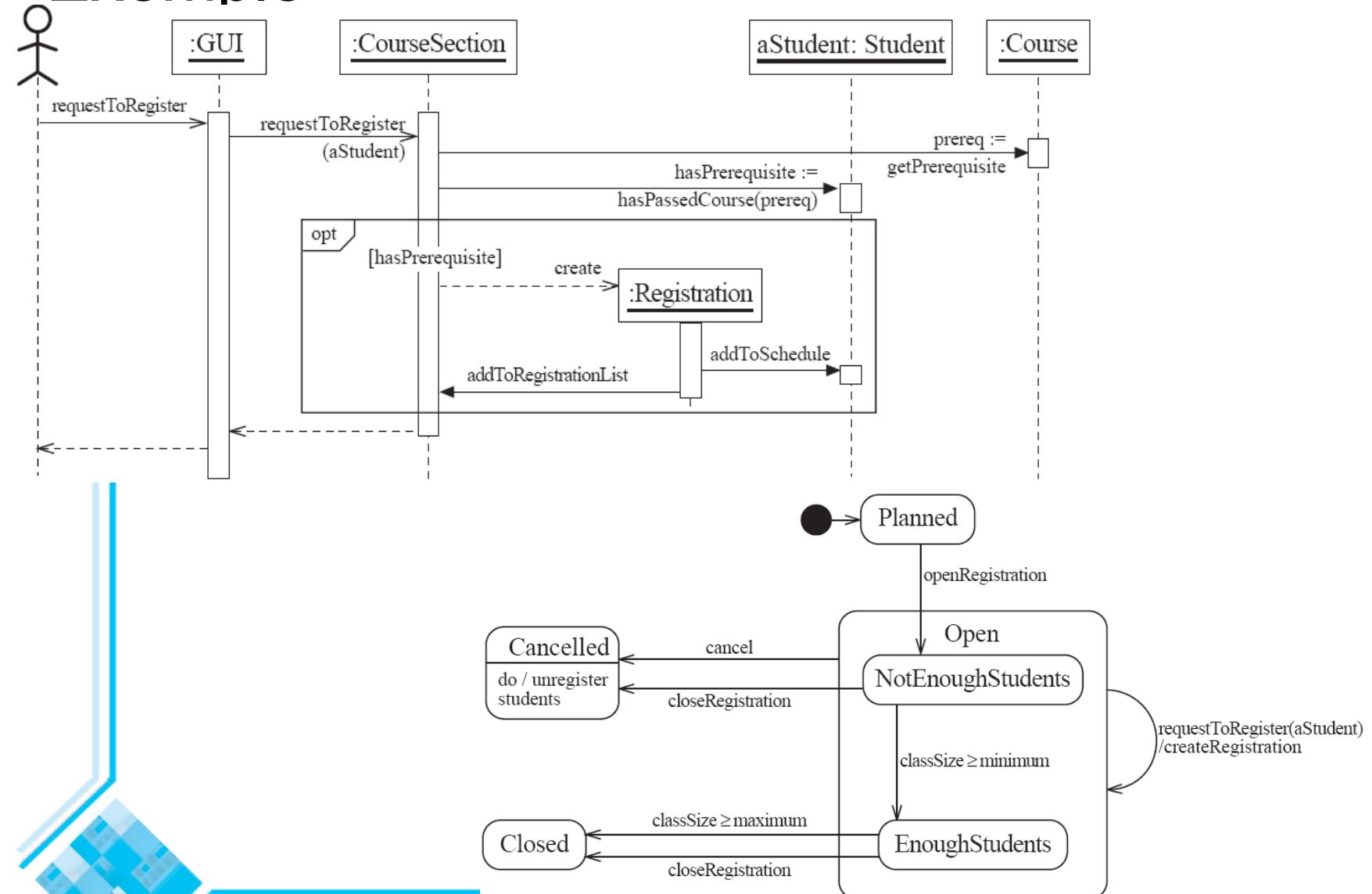
8.4 Implémenter une classe à l'aide de diagrammes d'activité et d'état

- Les diagrammes d'état et d'activité sont utilisés pour les éléments plus complexes du système
 - Pas toutes les classes justifient l'utilisation de tels diagrammes
- L'utilisation conjointe de diagrammes d'interaction, d'activité et d'état peuvent aider à concevoir la bonne implémentation.
- Ceci est particulièrement vrai lorsque le comportement d'une classe, d'un sous-système est distribué parmi plusieurs cas d'utilisation
- Un diagramme d'état est utile lorsque différentes conditions produisent une réponse différentes aux même événements

Un autre exemple



Exemple



Exemple: La classe CourseSection

États:

- ‘Planned’:
closedOrCancelled == false && open == false
- ‘Cancelled’:
closedOrCancelled == true &&
registrationList.size() == 0
- ‘Closed’ (course section is too full, or being taught):
closedOrCancelled == true &&
registrationList.size() > 0

Exemple: La classe CourseSection

États:

- ‘Open’ (accepting registrations):

open == true

- ‘NotEnoughStudents’ (substate of ‘Open’):

open == true &&

registrationList.size() < course.getMinimum()

- ‘EnoughStudents’ (substate of ‘Open’):

open == true &&

registrationList.size() >= course.getMinimum()

Exemple: La classe CourseSection

```
public class CourseSection
{
    // The many-1 abstraction-occurrence association (Figure 8.2)
    private Course course;

    // The 1-many association to class Registration (Figure 8.2)
    private List registrationList;

    // The following are present only to determine the state
    // (as in Figure 8.19). The initial state is 'Planned'
    private boolean open = false;
    private boolean closedOrCanceled = false;
```

Exemple: La classe CourseSection

```
public CourseSection(Course course)
{
    this.course = course;
    registrationList = new LinkedList();
}

public void openRegistration()
{
    if(!closedOrCanceled) // must be in 'Planned' state
    {
        open = true; // to 'OpenNotEnoughStudents' state
    }
}
```

Exemple: La classe CourseSection

```
public void closeRegistration()
{
    // to 'Canceled' or 'Closed' state
    open = false;
    closedOrCanceled = true;
    if (registrationList.size() < course.getMinimum())
    {
        unregisterStudents(); // to 'Canceled' state
    }
}

public void cancel()
{
    // to 'Canceled' state
    open = false;
    closedOrCanceled = true;
    unregisterStudents();
}
```

Exemple: La classe CourseSection

```
public void requestToRegister(Student student)
{
    if (open) // must be in one of the two 'Open' states
    {
        // The interaction specified in the sequence diagram of Figure 8.4
        Course prereq = course.getPrerequisite();
        if (student.hasPassedCourse(prereq))
        {
            // Indirectly calls addToRegistrationList
            new Registration(this, student);
        }
        // Check for automatic transition to 'Closed' state
        if (registrationList.size() >= course.getMaximum())
        {
            // to 'Closed' state
            open = false;
            closedOrCanceled = true;
        }
    }
}
```

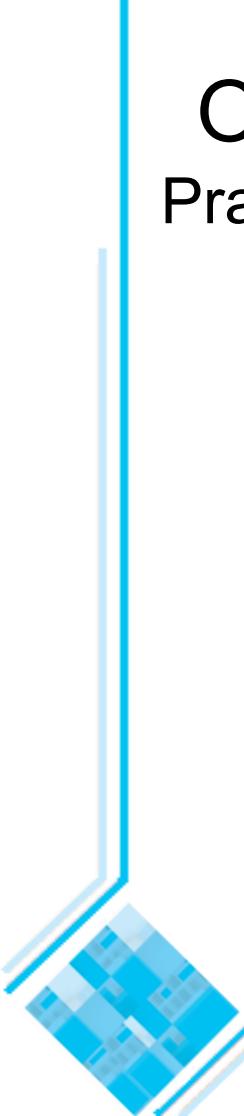
Exemple: La classe CourseSection

```
// Private method to remove all registrations
// Activity associated with 'Canceled' state.
private void unregisterStudents()
{
    Iterator it = registrationList.iterator();
    while (it.hasNext())
    {
        Registration r = (Registration)it.next();
        r.unregisterStudent();
        it.remove();
    }
}
// Called within this package only, by the constructor of
// Registration to ensure the link is bi-directional
void addToRegistrationList(Registration newRegistration)
{
    registrationList.add(newRegistration);
}
```

8.5 Risques et difficultés dans la modélisation des interactions et du comportement

La modélisation dynamique est une tâche difficile

- Dans un système de grande taille, il existe de nombreux chemins que le système peut emprunter.
- Il est difficile d'attribuer le bon comportement aux bonnes classes:
 - Le processus de modélisation devrait être supervisé par le développeur le plus expérimenté*
 - Tous les aspects du modèle devrait être révisés attentivement.*
 - Travailler de façon itérative:*
 - *Développer un premier diagramme de classes, puis les cas d'utilisations, les responsabilités, et ensuite les diagrammes d'interaction, d'activité et d'état;*
 - *Réviser et modifier ensuite les diagramme afin de les rendre consistants*
 - Le fait de dessiner différents diagrammes représentant des aspects différents, mais très liés, aide à mettre en lumière les problèmes potentiels*



Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 9A:

Architecting and Designing Software

(Software Design)

9.1 Le Design

Définition:

- *Le Design* est un processus de résolution de problèmes dont l'objectif est d'identifier la meilleure façon de:
 - implémenter les exigences fonctionnelles d'un système...
 - tout en respectant les contraintes imposées par les exigences non-fonctionnelles (*ex: exigences de qualité, plateforme, processus*)
 - y incluant les contraintes de budget
 - et en adhérant à des principes de base menant à un logiciel de bonne qualité

Le design en tant que série de décisions

Tout designer doit faire face à une série de problèmes

- Ce sont des sous-problèmes devant être résolus.
- Chacun de ces sous-problèmes peut être résolu de différentes façons:
 - Ce sont des *options de design*.
- Le designer doit donc prendre des décisions de design afin de résoudre chacun de ces sous-problèmes.
 - Il faut donc être en mesure de toujours choisir la meilleure alternative.

Prendre des décisions

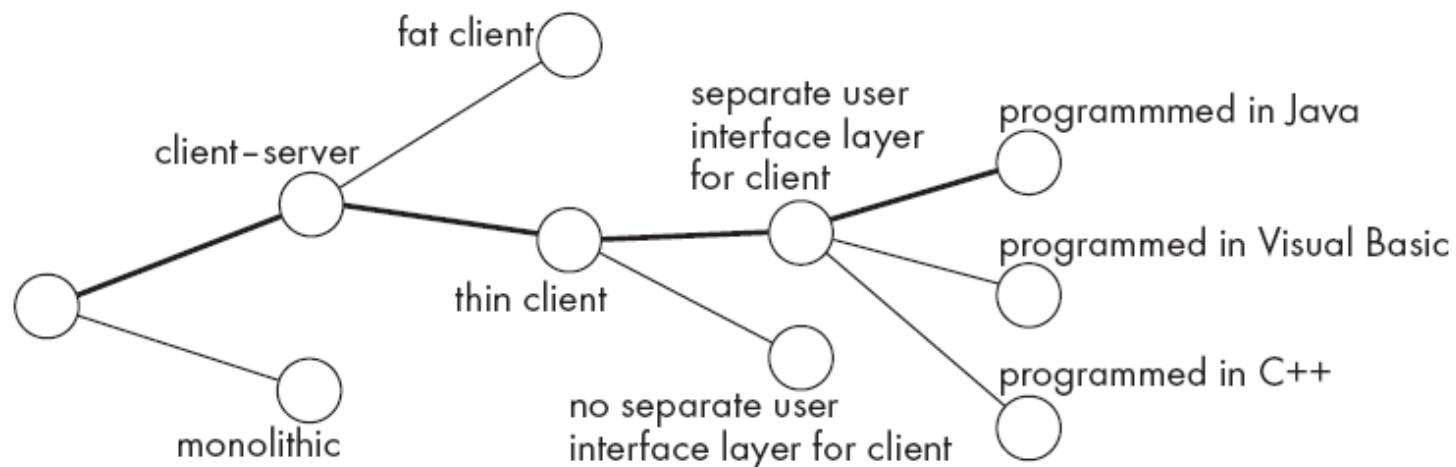
Afin de prendre une décision de design, l'ingénieur logiciel doit tenir compte:

- des exigences
- du design courant
- de la technologie disponible
- des principes de bon design
- ce qui a bien fonctionné dans le passé

L'espace de Design

L'espace engendré par l'ensemble des solutions possibles pour un design s'appelle l'espace de design

- Par exemple:



Composante

Tout élément logiciel ou matériel ayant un rôle bien défini.

- Une composante peut être isolée, et remplacée par une autre composante ayant des fonctionnalités équivalentes.
- De nombreux composants sont conçus pour être réutilisables.
- À l'inverse, d'autres remplissent des fonctions spéciales.

Module

Une composante définie au niveau de langage de programmation

- Par exemple des méthodes, des classes, des paquetages sont des modules en Java.

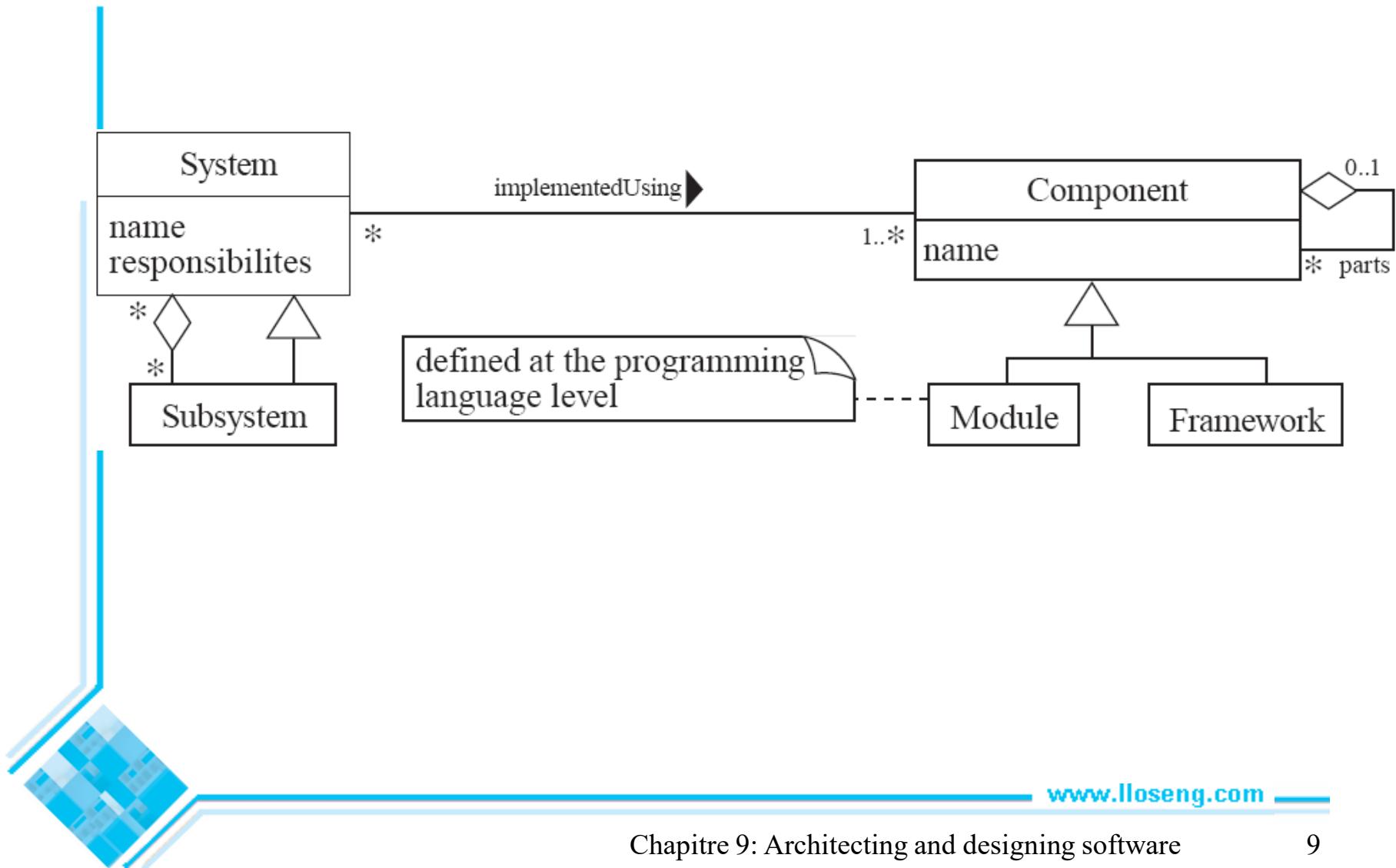


Système

Une entité logique matérielle, logicielle, ou les deux, et ayant un ensemble de responsabilités définies

- Un système est implémenté à l'aide d'un ensemble de composantes
- Un système continu à exister même après que ses composantes ont été changées ou remplacées.
- L'analyse des exigences consiste à déterminer les responsabilités d'un système.
- **Sous-système:**
 - Un système faisant partie d'un système plus grand et ayant une interface bien définie

Diagramme UML représentant les parties d'un système



Effectuer un design de haut en bas

De haut en bas

- Concevez d'abord la structure de très haut niveau du système.
- Ensuite, travaillez progressivement vers des décisions détaillées sur les constructions de bas niveau.
- Arrivez enfin à des décisions détaillées telles que:
 - le format de données;
 - les algorithmes qui seront utilisés.

Effectuer un design de bas en haut

De bas en haut

- Prenez des décisions sur les éléments **de bas niveau** (possiblement) **réutilisables**.
- Ensuite, **décidez comment ces éléments seront assemblés** pour créer des constructions de haut niveau.

Un mélange de ces deux approches est normalement utilisé:

- Une approche de haut en bas est nécessaire afin de garantir une bonne structure au système.
- Une approche de bas en haut est utile afin d'assurer que des composantes réutilisables seront conçues.

Différents aspects du design

- *Design d'architecture:*
 - La division du système en sous-systèmes et composantes
 - Comment ceux-ci seront interconnectées.
 - Comment vont-ils interagir.
 - Leurs interfaces.
- *Design de classes:*
 - Attributs, opérations et associations.
- *Design de l'interface utilisateur*
- *Design des algorithmes:*
 - Étude de leur efficacité.
- *Design de protocoles:*
 - Les messages et règles utilisés dans la communication.

9.2 Principes menant à un design de qualité

Objectifs généraux d'un bonne design:

- Accroître les profits par la réduction des coûts et l'accroissement des revenus
- S'assurer de l'adhérence aux exigences
- Accélérer le développement
- Accroître les attributs de qualité tels
 - Utilisabilité
 - Efficacité
 - Fiabilité
 - Maintenabilité
 - Réutilisabilité

Principe de design 1: Diviser pour régner

Afin de maîtriser un système complexe, il faut le subdiviser en une série de plus petits systèmes

- Différentes personnes peuvent ainsi travailler sur chacune des sous-parties.
- Chaque ingénieur peut se spécialiser
- Chacune des composantes est plus petite et plus facile à comprendre.
- Certaines parties peuvent être remplacées, changées ou modifiées sans avoir à modifier les autres parties du système.

Différentes façons de subdiviser un système

- Un système distribué est divisé en clients et serveurs
- Un système est divisé en sous-systèmes
- Un sous-système peut être subdivisé en paquetages
- Un paquetage est composé de classes
- Une classe est composée de méthodes

Principe de design 2: Accroître la cohésion autant que possible

Un système a une cohésion élevée si les éléments interreliés sont groupés ensemble et si les éléments indépendants sont dans des groupes distincts

- Le système devient alors plus facile à saisir et éventuellement a y apporter des changements
- Type de cohésion:
 - Fonctionnelle, en couches, communicationnelle, séquentielle, procédurale, temporelle, utilitaire

Cohésion fonctionnelle

Lorsque que tout le code effectuant le calcul d'un certain résultat se trouve au même endroit

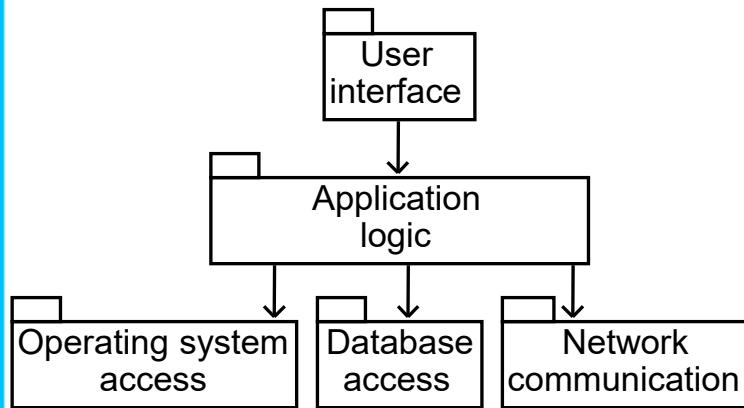
- i.e. lorsqu'un module effectue le calcul d'un seul résultat, sans effets secondaires.
- Bénéfices:
 - Facilite la compréhension
 - Plus facile à réutiliser
 - Plus facile à remplacer
- Un module gérant une base de données, créant des fichiers ou interagissant avec un utilisateur n'est pas fonctionnellement cohésif

Cohésion en couches

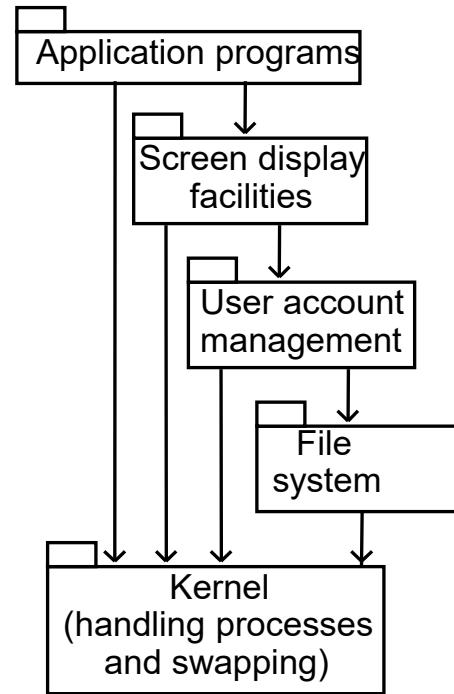
Tous les fournisseurs d'accès à un ensemble de services interreliés sont groupés ensemble

- Les différentes couches devraient former une hiérarchie
 - Les couches de plus haut niveaux peuvent accéder aux couches de plus bas niveaux,
 - Les couches de bas niveaux n'accèdent pas aux couches de plus haut niveau
- L'ensemble des procédures qu'une couche met à la disposition des autres couches pour accéder aux services qu'elle offre est *l'application programming interface (API)*
- Une couche peut être remplacée sans que cela n'affecte les autres couches
 - Il faut simplement reproduire le même API

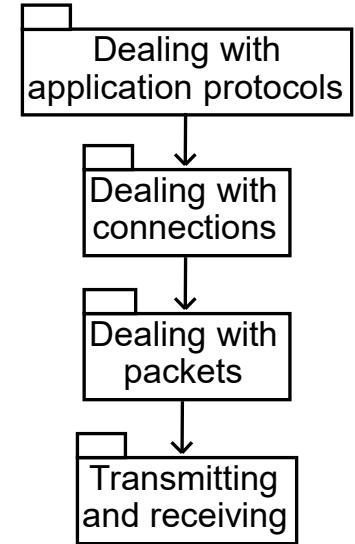
Exemples de systèmes en couches



a) Typical layers in an application program



b) Typical layers in an operating system



c) Simplified view of layers in a communication system

Cohésion communicationnelle

Tous les modules accédant ou manipulant les même données sont groupées ensemble

- Une classe a une bonne cohésion communicationnelle si
 - Toutes les opérations de manipulation de données sont contenus dans cette classe.
 - La classe ne gère que les données qui la concerne.
- Avantage:
 - Lorsqu'un changement doit être effectué sur les données, tout le code concerné se trouve au même endroit

Cohésion séquentielle

Les procédures pour lesquelles l'une produit une sortie servant d'entrée à une autre sont groupées ensemble

- Ce genre de cohésion est valable lorsque les autres types de cohésion ont été achevés.

Cohésion procédurale

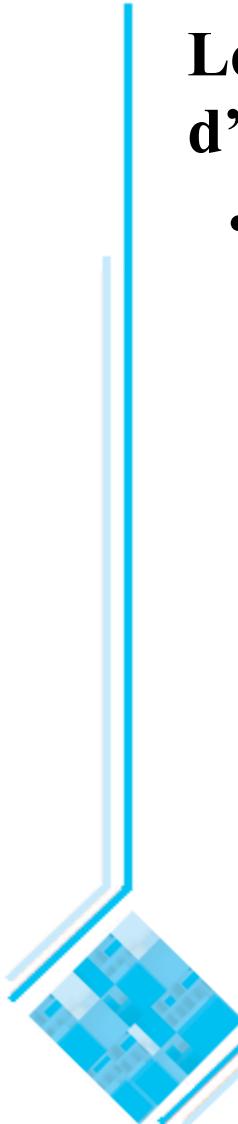
Les procédures qui se succèdent l'une après l'autre sont groupées ensemble

- Même si une ne produit pas un résultat utilisé par la suivante.
- Plus faible que la cohésion séquentielle.

Cohésion temporelle

Les opérations effectuées lors de la même phase d'exécution sont groupées ensemble

- Par exemple, tout le code utilisé lors de l'initialisation pourrait être regroupé.



Cohésion utilitaire

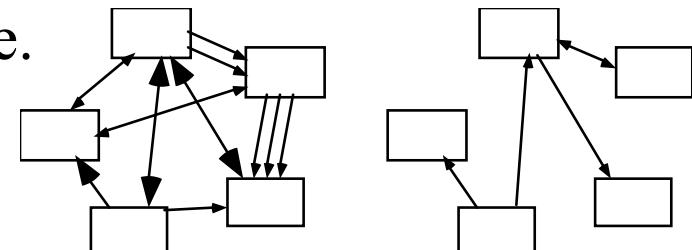
Les utilitaires interreliés sont rassemblés lorsqu'il n'y a aucun moyen de les regrouper en utilisant une forme de cohésion plus forte

- Un utilitaire est une procédure d'intérêt général pouvant être utilisé dans une grande variété d'applications.
- Les utilitaires sont hautement réutilisables
- Par exemple, la classe **java.lang.Math**.

Principe de design 3: Réduire le couplage en autant que possible

Il y a *couplage* lorsqu'une interdépendance existe entre deux modules

- Lorsqu'il y a dépendance, un changement dans une composante implique un changement dans une autre composante.
- Un réseau complexe de dépendances rend difficile la compréhension d'une composante.
- Type de couplage:
 - Contenu, Commun, Contrôle, Estampillage, Données, Appel, Type, Inclusion, Externe



Couplage de contenu

Lorsqu'une composante *subrepticement* modifie les données internes d'une autre composante

- Le fait d'encapsuler les données réduit considérablement le couplage
 - Elles sont déclarées private
 - Avec des méthodes get et set

Exemple de couplage de contenu

```
public class Line
{
    private Point start, end;
    ...
    public Point getStart() { return start; }
    public Point getEnd()   { return end; }
}

public class Arch
{
    private Line baseline;
    ...
    void slant(int newY)
    {
        Point theEnd = baseline.getEnd();
        theEnd.setLocation(theEnd.getX(),newY);
    }
}
```

Couplage commun

Lorsqu'une variable globale est utilisée

- Toutes les composantes utilisant cette variable globale deviennent alors couplées les unes aux autres
- Une forme plus faible de couplage est présente lorsque la variable est accessible à un nombre restreint de classes
 - e.g. un paquetage Java
- Acceptable lorsque la variable globale fait référence à des paramètres globaux du système
- Le Singleton est un moyen d'offrir un accès contrôlé à un objet

Couplage de contrôle

Lorsqu'une procédure en appelle une autre en utilisant une variable de contrôle ou une commande contrôlant l'exécution de la procédure appelée

- Afin d'effectuer un changement, il faut modifier à la fois l'appelé et l'appelant
- L'utilisation d'une opération polymorphique constitue la meilleure façon d'éviter le couplage de contrôle
- Une autre façon de réduire ce type de couplage consiste à avoir recours à une table *look-up*
 - Chaque commande est alors associée une méthode qui sera appelée lorsque ces commande est lancée

Exemple de couplage de contrôle

```
public routineX(String command)
{
    if (command.equals("drawCircle"))
    {
        drawCircle();
    }
    else
    {
        drawRectangle();
    }
}
```

Couplage d'estampillage

Lorsqu'une classe est déclarée dans la liste des arguments d'une méthode

- Une classe en utilise donc une autre
 - Afin de réutiliser une classe, il faut aussi utiliser l'autre
- Pour réduire ce type de couplage
 - Utiliser une interface
 - Transmettre que des variables simples

Exemple de couplage d'estampillage

```
public class Emailer  
{  
    public void sendEmail(Employee e, String text)  
    {...}  
    ...  
}
```

En utilisant des données simples:

```
public class Emailer  
{  
    public void sendEmail(String name, String email, String text)  
    {...}  
    ...  
}
```

Exemple de couplage d'estampillage

En utilisant une interface:

```
public interface Addressee
{
    public abstract String getName();
    public abstract String getEmail();
}

public class Employee implements Addressee {...}

public class Emailer
{
    public void sendEmail(Addressee e, String text)
    {...}
    ...
}
```

Couplage de données

Lorsque les types d'arguments de méthode sont des primitives ou classes simples (par exemple, String)

- Plus il y a d'argument, plus ce couplage est fort
 - Les méthodes appelantes doivent fournir tous ces arguments
- Il faut réduire ce type de couplage en évitant d'utiliser des arguments non-nécessaires
- Il y a souvent un compromis à faire entre couplage de données et couplage d'estampillage
 - i.e. réduire l'un accroît l'autre

Couplage d'appel

Lorsqu'une méthode en appelle une autre

- Ces méthodes sont couplée car le comportement de l'une dépend du comportement de l'autre
- Il y aura toujours du couplage d'appel dans tout système
- Si une même séquence d'appel se répète fréquemment, alors il faudrait songer à encapsuler cette séquence en créant une méthode regroupant ces appels

Couplage de type

Lorsqu'un module utilise un type défini dans un autre module

- Est présent chaque fois qu'une classe déclare un attribut d'une autre classe.
- La conséquence est que si le type (la classe) est modifié, alors la classe qui en fait usage devra aussi être changé
- Toujours utiliser le type le plus général

Couplage d'inclusion

Lorsqu'une composante en importe une autre

- (un paquetage en Java)

ou en inclut une autre

- (comme en C++).

- La composante qui procède à l'inclusion devient dépendante de la composante inclue.

- Si cette composante incluse est modifiée ou si quelque chose y est ajouté.

—Il peut se produire un conflit; un nouvel élément pouvant avoir le même nom qu'un élément existant.

Couplage externe

Lorsqu'un module dépend d'une librairie, d'un système d'exploitation, d'un matériel

- Il faut réduire au maximum la dispersion de cette dépendance à travers le code.
- La Façade est un moyen efficace de réduire ce type de couplage

Principe de design 4: Maintenir le niveau d'abstraction aussi élevé que possible

Assurez-vous que vos conceptions vous permettent de masquer ou de différer la considération des détails, réduisant ainsi la complexité

- Une bonne abstraction utilise toujours le principe de *masquage de l'information*
- Une abstraction permet de saisir l'essence d'un système sans avoir à en connaître les détails de son implémentation
- Exemples d'abstractions:
 - Classes
 - Associations de UML
 - Interfaces
 - Machines d'état

Principe de design 5: Accroître la réutilisabilité en autant que possible

Concevoir le design de façon à ce que les différents aspects du système soient utilisable dans différents contextes

- Généraliser le design en autant que possible
- Simplifier le design en autant que possible
- Ajouter des options aux différents modules

Principe de design 6: Réutiliser autant de composantes que possible

La réutilisation est le principe complémentaire au principe de réutilisabilité

- Réutiliser les designs existants permet de profiter de l'effort investi par les concepteurs de composantes réutilisables
 - Le clonage* ne doit pas être vu comme une forme de réutilisation

Principe de design 7: Rendre le design flexible

Anticiper les changements qui seront apportés au design

- Réduire le couplage et accroître la cohésion
- Créer des abstractions
- Pas de *hard-coding*
- Laisser toutes les options ouvertes
 - Ne pas restreindre inutilement les options
- Utiliser du code réutilisable et rendre le code réutilisable

Principe de design 8: Anticiper l'obsolescence

Planifier les changements de technologies ou d'environnements de façon à ce que peu ou pas de modifications soient requises

- Éviter d'utiliser des versions préliminaires de nouvelles technologies
- Éviter d'utiliser des bibliothèques spécifiques à des environnements particuliers
- Éviter d'utiliser des éléments mal documentés
- Éviter d'utiliser des logiciels ou du matériel provenant de fournisseurs peu fiables
- Utiliser des technologies standards supportées par de multiples vendeurs

Principe de design 9: Concevoir des designs portables

Permettre au logiciel de s'exécuter sur autant de plate-formes que possible

- Éviter d'utiliser des éléments spécifiques à un environnement particulier
- E.g. une librairie disponible seulement sous Microsoft Windows



Principe de design 10: Concevoir un design ais      tester

Faciliter l'incorporation de tests

- Concevoir un programme permettant de tester automatiquement le logiciel
 - S'assurer que toutes les fonctionnalit  s du logiciel peuvent   tre conduites par un programme externe sans avoir    passer par l'interface utilisateur
- Sous Java, il est possible de cr  er une m  thode `main()` dont le r  le est de tester les autres m  thodes de la classe
- Utilisez Junit ou des frameworks similaires

Principe de design 11: Concevoir un design défensif

Ne jamais faire confiance aux composantes utilisées

- Traiter tous les cas possibles, même les cas d'usage inappropriés
- Vérifier que toutes les données en entrée sont valides:
les *préconditions*
 - Toutefois un design trop précautionneux peu résulter en une multiplication inefficace de séquences répétées vérifications

Design par contrats

Il s'agit d'une technique permettant de concevoir de façon systématique un design défensif qui demeure efficace

- Principe de base
 - Chaque méthode conclut un *contrat* avec ses appelants
- Ce contrat introduit un ensemble d'assertions définissant:
 - Quelles sont les *préconditions* requises pour que la méthode puisse démarrer son exécution
 - Quelles sont les *postconditions* que la méthode assure rencontrer à la fin de son exécution
 - Quels sont les *invariants* que la méthode requiert et garantie au cours de son exécution

9.3 Des techniques permettant de prendre les bonnes décision de design

Définir les priorités et les objectifs afin de choisir les bonnes alternatives

- Étape 1: Lister et décrire les différentes alternatives possibles.
- Étape 2: Lister les avantages et désavantages de chacune des alternatives en fonction des priorités et objectifs.
- Étape 3: Déterminer quelles alternatives ne permettent pas de rencontrer certains objectifs.
- Étape 4: Choisir l'alternative qui permet le mieux de rencontrer les objectifs définis.
- Étape 5: Ajuster les priorités pour les décision subséquentes.

Exemple de priorités et objectifs

Imaginer un système ayant les objectifs suivants (par ordre de priorité):

- **Sécurité:** Le code cryptographique ne doit pas être brisable en moins de 10,000 heures de CPU sur un processeur Intel de 5.3GHz et 10 cœurs en utilisant les techniques connues de crypto-analyse.
- **Efficacité du CPU.** Temps de réponse inférieur à 200 ms.
- **Largeur de bande:** Pas plus de 500KB de données par transaction.
- **Mémoire.** Pas plus de 80MB de RAM.
- **Portabilité.** Doit être exécutable sous Windows 10, macOS Big Sur, et Linux
- **Maintenabilité.** Pas d'objectif spécifique.

Exemple d'évaluation d'alternatives

	<i>Security</i>	<i>Maintainability</i>	<i>Memory efficiency</i>	<i>CPU efficiency</i>	<i>Bandwidth efficiency</i>	<i>Portability</i>
Algorithm A	High	Medium	High	Medium; DNMO	Low	Low
Algorithm B	High	High	Low	Medium; DNMO	Medium	Low
Algorithm C	High	High	High	Low; DNMO	High	Low
Algorithm D	—	—	—	Medium; DNMO	DNMO	—
Algorithm E	DNMO	—	—	Low; DNMO	—	—

‘DNMO’signifie: “ne rencontre pas cet objectif”

Analyse coût-bénéfice afin de déterminer l'alternative la plus avantageuse

- Le coût est constitué de:
 - Le coût incrémental lié à l'ingénierie du logiciel envisagé, y incluant sa maintenance
 - Le coût incrémental du développement de la technologie envisagée
 - Le coût incrémental subit par les utilisateur et l'équipe de support
- Les bénéfices sont constitué de:
 - La réduction du temps de développement présent et futur
 - L'accroissement anticipé des ventes ou de bénéfices financiers profitables aux utilisateurs

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapitre 9B:

Architecting and Designing Software

(Software Architecture)

9.4 Architecture logicielle

L'architecture logicielle est le processus de conception de l'organisation globale du système et incluant:

- La subdivision du logiciel en sous-systèmes.
- Les décisions à prendre concernant leur interactions.
- La détermination des interfaces.
 - L'architecture est le cœur du système, tous les ingénieurs impliqués doivent donc bien la maîtriser.
 - L'architecture va souvent contraindre l'efficacité, la réutilisabilité et la maintenabilité du système.

L'importance d'une bonne architecture logicielle

Pourquoi développer un bon modèle de l'architecture:

- Afin que tous aient une meilleure compréhension du système
- Afin que les différents concepteurs puissent travailler isolément sur différentes composantes du système
- Afin d'être mieux préparer à étendre les fonctionnalités du système
- Afin de faciliter la réutilisation et la réutilisabilité

Un bon modèle architectural

Une architecture sera habituellement décrite sous divers points de vue

- La subdivision du système en sous-systèmes
- Les interfaces des différents sous-systèmes
- La dynamique de l'interaction requise entre les différentes composantes du système
- Les données utilisées et échangées à travers le système
- Les composantes qui existeront et leur répartition physique

Concevoir une architecture stable

Afin d'assurer la maintenabilité et la fiabilité du système, une architecture doit demeurer stable.

- Une architecture est stable lorsque l'ajout d'un nouvel élément ne requiert que des changements mineurs



Le développement d'un modèle de l'architecture

Commencer avec une première ébauche de l'architecture globale

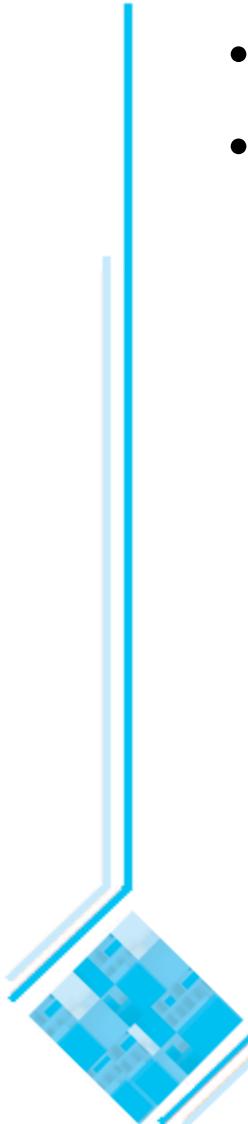
- Basée sur les exigences et cas d'utilisation principales
- Déterminer les composantes principales
- Sélectionner certains patrons architecturaux
 - On va les voir plus tard
- *Suggestion:* plusieurs équipes distinctes peuvent travailler de façon indépendante pour ensuite comparer et fusionner leurs idées

Le développement d'un modèle de l'architecture

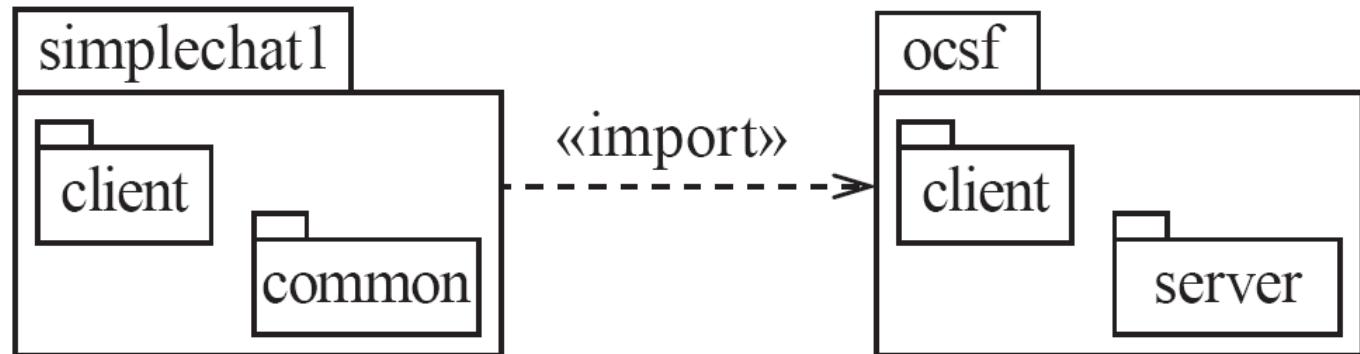
- Raffiner l'architecture
 - Identifier les interactions principales entre composantes et les interfaces qui seront requises
 - Décider comment les données et les procédures seront réparties parmi les composantes
 - Déterminer si certains éléments seront réutilisés ou si, par exemple, un framework pourrait être conçu
- Considérer tour à tour chacun des cas d'utilisation afin de s'assurer qu'ils seront réalisables
- Faire évoluer cette architecture

La description d'une architecture avec UML

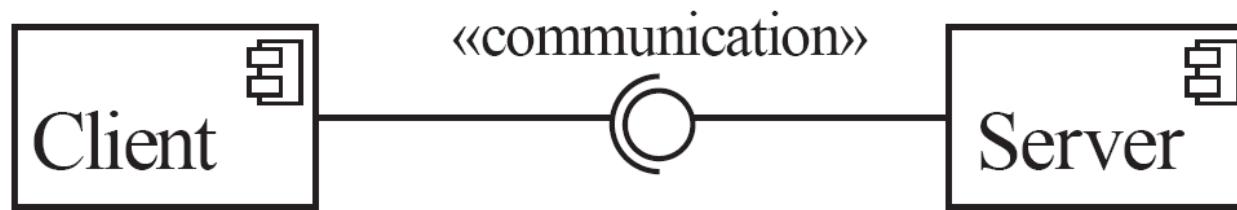
- Tous les diagrammes UML peuvent être utilisés
- Trois diagrammes sont particulièrement utiles:
 - Diagrammes de paquetage
 - Diagramme de composante
 - Diagramme de déploiement



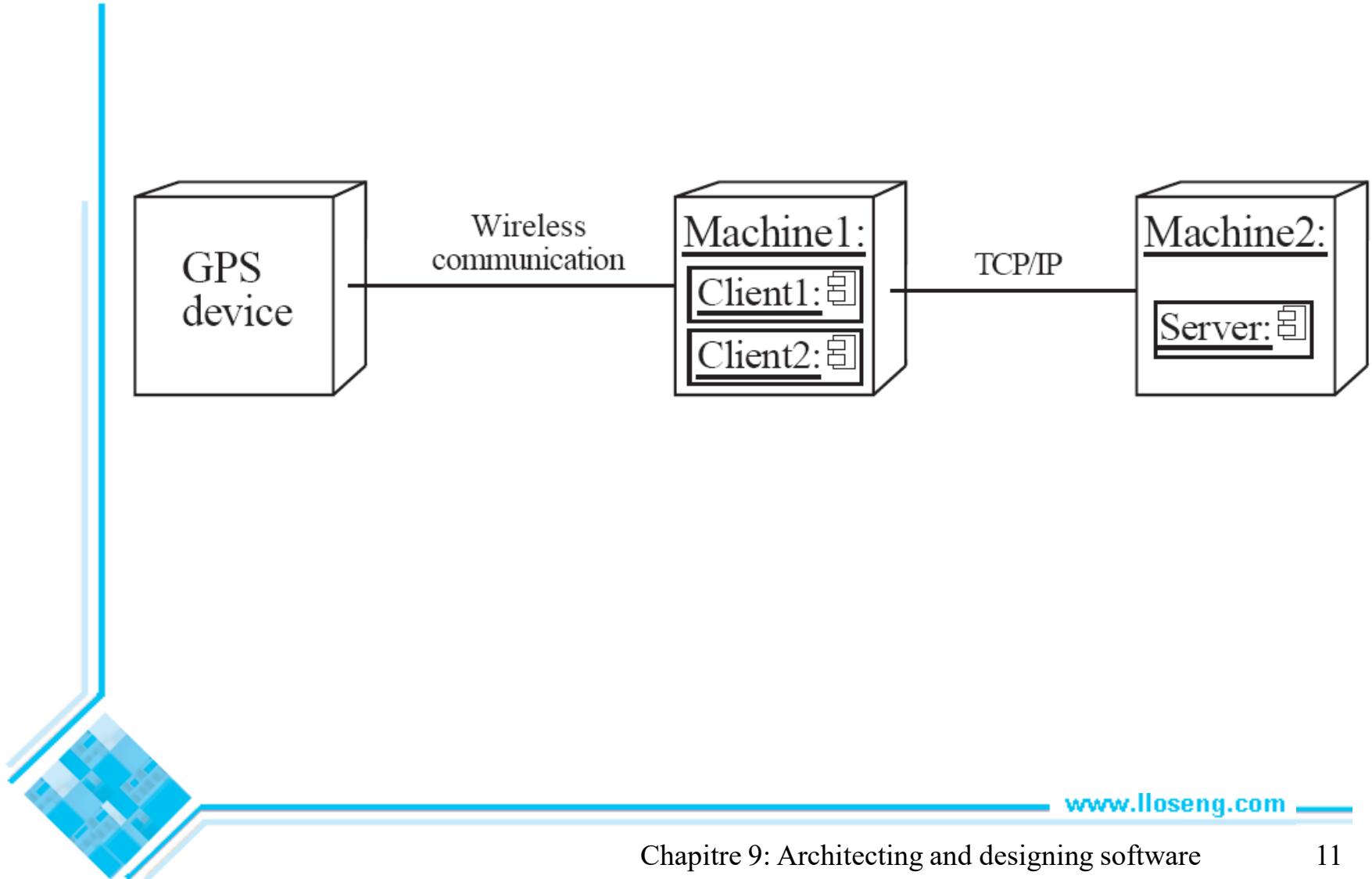
Diagrammes de paquetage



Diagrammes de composantes



Diagrammes de déploiement



9.5 Patrons architecturaux

La notion de *patron* peut aussi s'appliquer à l'architecture.

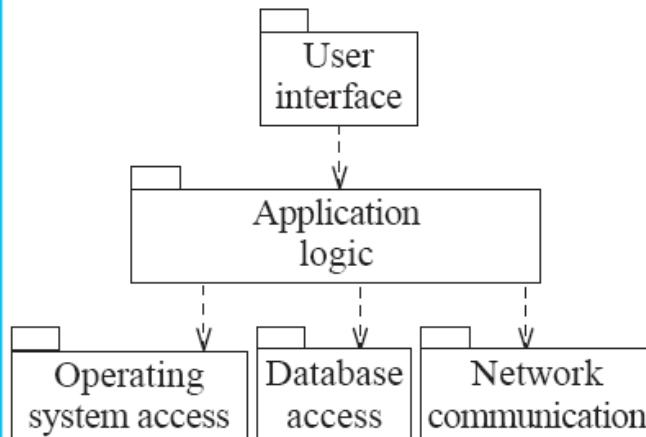
- Ce sont donc des *patrons architecturaux* ou *styles d'architecture*.
- Ceux-ci permettent de concevoir une architecture solide et flexible en utilisant les composantes appropriées
 - Ces composantes doivent être indépendantes les une des autres, aussitôt que possible.

L'architecture multi-couches

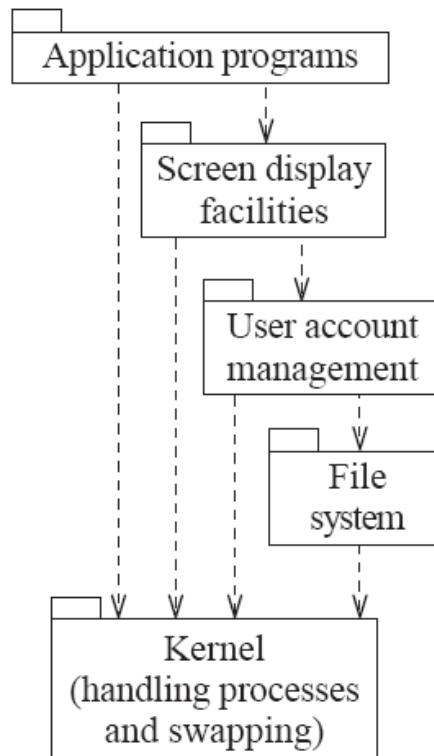
Dans un système en couches, chaque couche ne communique qu'avec la couche inférieure.

- Chacune des couches doit avoir une interface clairement définie.
 - Les couches inférieures offrent des services aux couches supérieures.
- Un système complexe se construit en superposant des couches de niveaux d'abstraction croissant.
 - L'interface utilisateur est l'une de ces couches distinctes.
 - La couche immédiatement sous l'interface utilisateur offre les services définis par les cas d'utilisation.
 - Les couches inférieures offre des services plus spécifiques.
 - e.g. accès à des bases de données

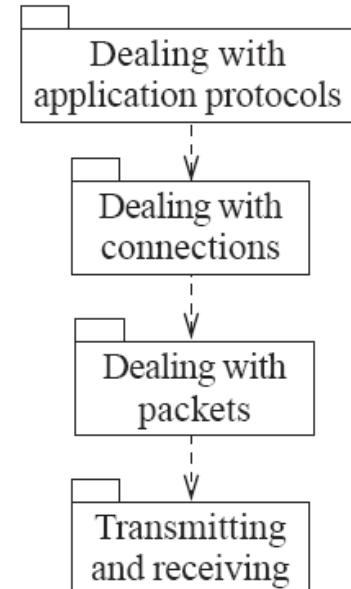
Exemple d'un système multi-couches



(a) Typical layers in an application program



(b) Typical layers in an operating system

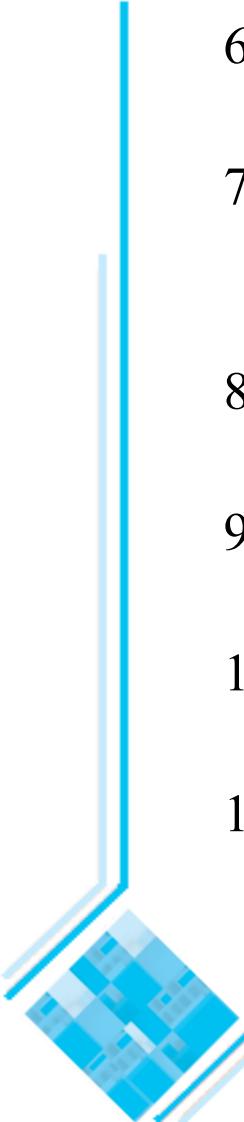


(c) Simplified view of layers in a communication system

L'architecture multi-couche et les principes de design

1. *Diviser pour régner*: les couches sont conçues indépendamment.
2. *Accroître la cohésion*: les couches présentent une cohésion en couches.
3. *Réduire le couplage*: les couches inférieures ne connaissent rien des couches supérieures et communiquent entre elles via leur API.
4. *Accroître l'abstraction*: il n'est pas nécessaire de connaître les détails internes des couches inférieures pour concevoir les couches supérieures.
5. *Accroître la réutilisabilité*: les couches inférieures peuvent être conçues de façon à offrir des services génériques.

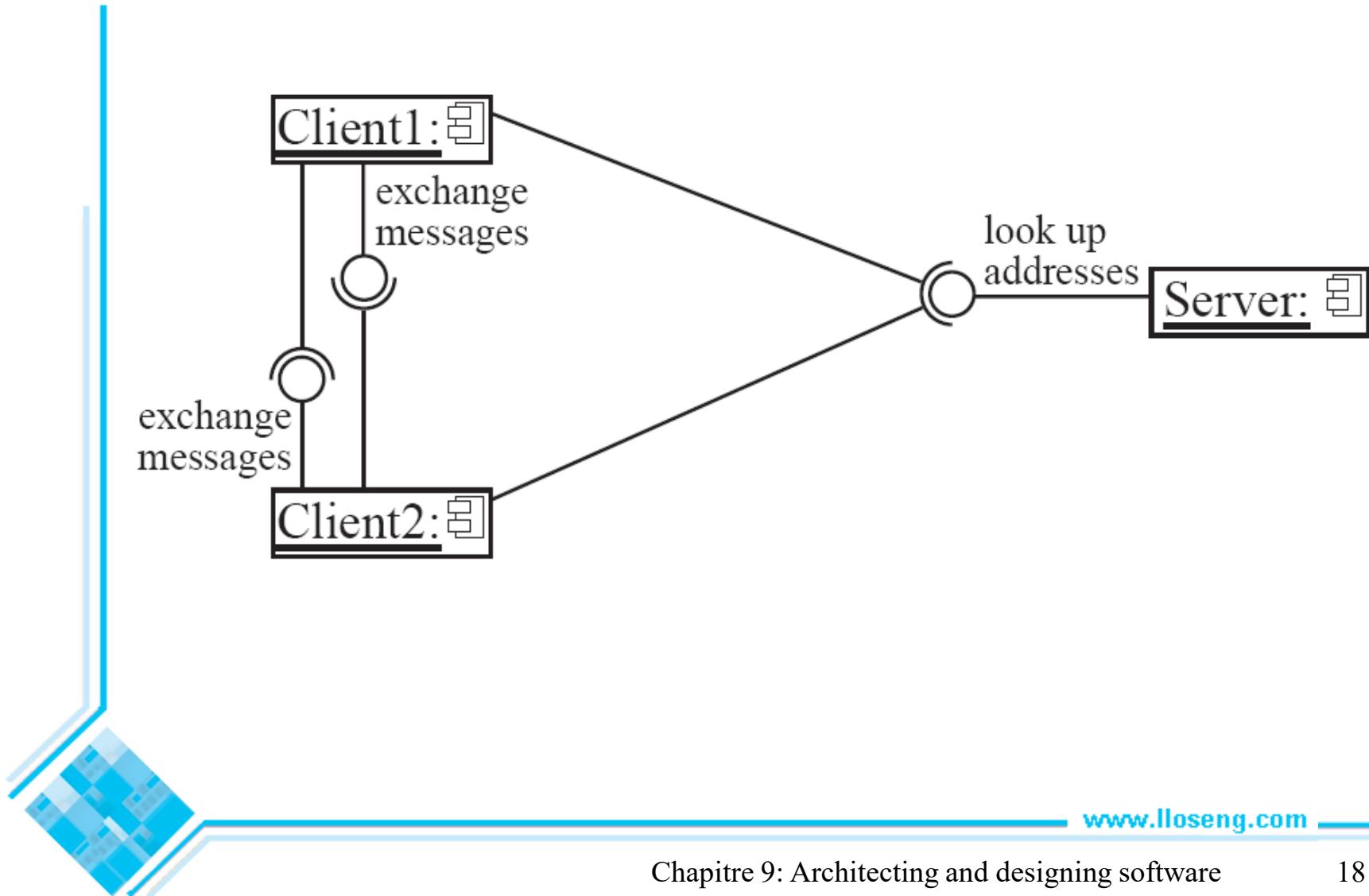
L'architecture multi-couche et les principes de design

- 
6. *Accroître la réutilisation*: toute couche existante offrant les services requis peut être utilisée.
 7. *Accroître la flexibilité*: vous pouvez ajouter de nouvelles fonctionnalités basées sur des services de niveau inférieur ou remplacer des couches de niveau supérieur.
 8. *Anticiper l'obsolescence*: en isolant les composantes, elles deviennent plus résistantes à l'obsolescence.
 9. *Concevoir des designs portables*: toutes les fonctionnalités dépendantes peuvent être confinées aux couches inférieures.
 10. *Faciliter les tests*: chacune des couches peut être testée indépendamment.
 11. *Concevoir de façon défensive*: l'API est l'endroit idéal pour placer les assertions de vérification.

L'architecture client-serveur et autres architectures distribuées

- Au moins une des composantes joue le rôle de *serveur* pouvant recevoir des connexions.
- Au moins une des composantes joue le rôle de *client* pouvant se connecter à un serveur et requérir des services.
- L'extension de cette architecture est le patron pair-à-pair.
 - Un système composé des plusieurs composantes distribué à travers plusieurs hôtes.

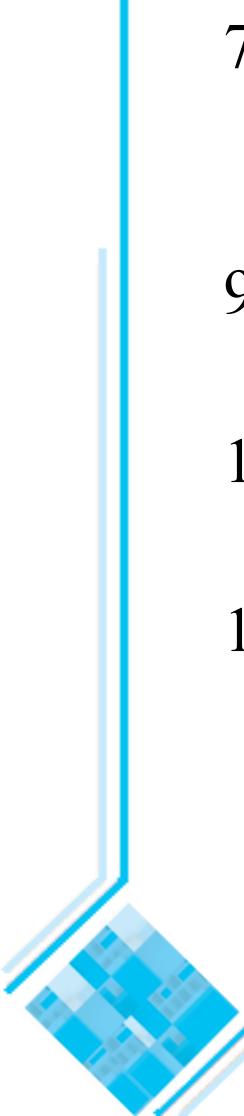
Un exemple de systèmes distribués



L'architecture distribuée et les principes de design

1. *Diviser pour régner*: la division du système en clients et serveurs est une manière très efficace de subdiviser le système.
— Chacune de ces composantes peut être développé séparément.
2. *Accroître la cohésion*: le serveur offre des services cohésifs aux clients.
3. *Réduire le couplage*: il existe généralement un seul canal de communication.
4. *Accroître l'abstraction*: chacune des composantes distribuées constituent de bonne abstractions.
6. *Accroître la réutilisation*: il est souvent possible de trouver des composantes à réutiliser sur lesquelles construire un bon système distribué

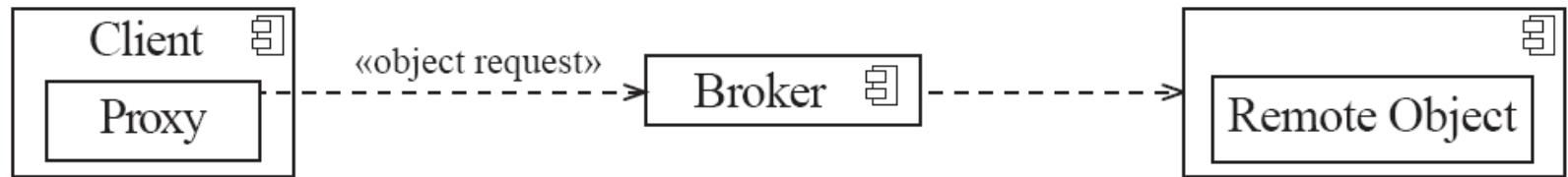
L'architecture distribuée et les principes de design

- 
7. *Accroître la flexibilité*: Les systèmes distribués peuvent souvent être facilement reconfigurés en ajoutant des serveurs ou des clients supplémentaires.
 9. *Concevoir des designs portables*: des clients sur différentes plateformes peuvent communiquer entre eux.
 - 10 *Faciliter les tests*: les client et serveurs peuvent être testés de façon indépendante.
 11. *Concevoir de façon défensive*: les vérifications sont insérés dans le traitement des messages.

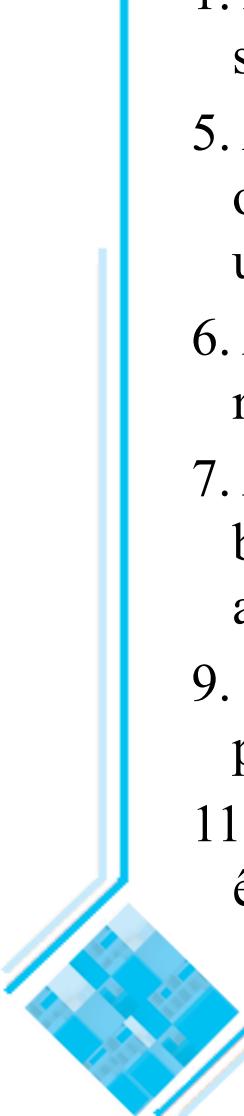
L'architecture courtier (*broker*)

- Le but est de distribuer de façon transparente différents aspects du système
 - Un objet fait appel aux méthodes d'un autre objet sans savoir que celui-ci est localisé sur une autre machine.
 - CORBA est l'un des standards bien connu permettant de construire ce genre d'architecture.

Exemple d'une architecture courtier



L'architecture courtier et les principes de design

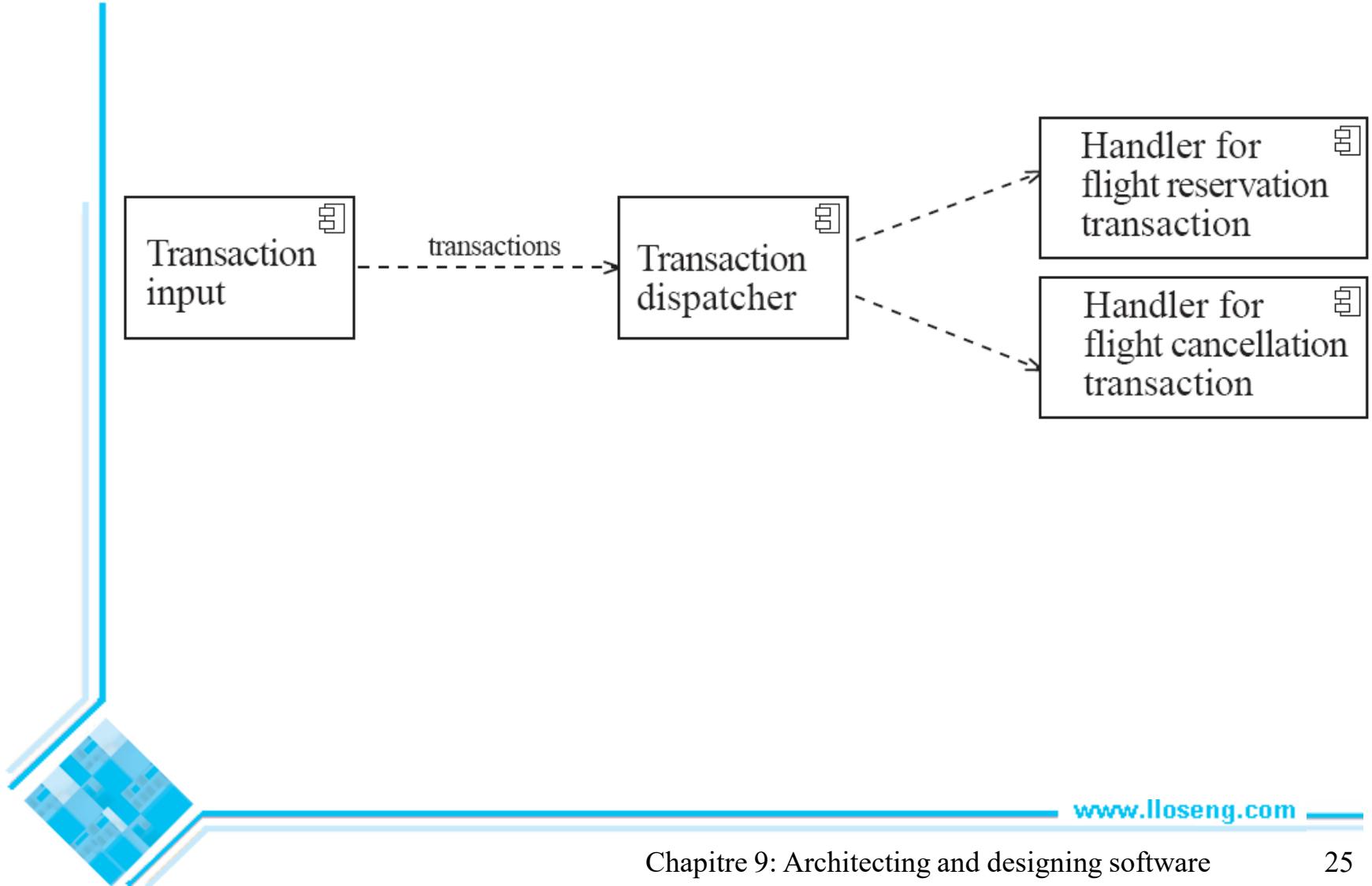
- 
1. *Diviser pour régner*: les objets distribués peuvent être conçus séparément.
 5. *Accroître la réutilisabilité*: il est souvent possible de concevoir les objets distribués de façon à ce que les objets distribués puisse être utilisés par d'autres systèmes.
 6. *Accroître la réutilisation*: différents objets distribués peuvent être réutilisé.
 7. *Accroître la flexibilité*: Le courtier peut être mis à jour selon les besoins ou vous pouvez rediriger le proxy pour qu'il communique avec un autre courtier.
 9. *Concevoir des designs portables*: les différents objets en transaction peuvent être localisés sur des plateformes différentes.
 11. *Concevoir de façon défensive*: des assertions peuvent facilement être ajoutés aux objets distribués

L'architecture transactionnelle

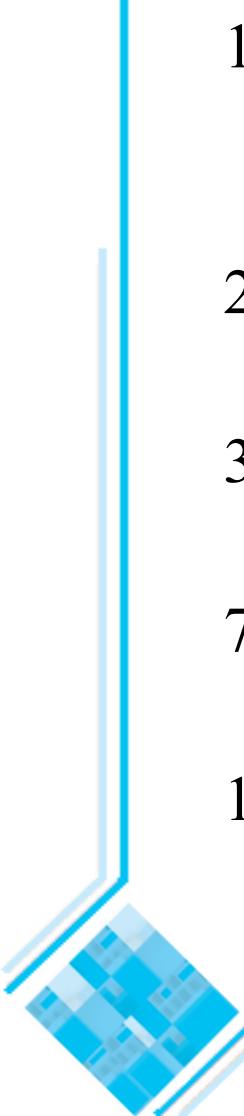
Un processus traite les différentes entrées une à une.

- Chaque entrée décrit une *transaction* – une commande qui modifie les données stockées dans le système
- Il existe un composant de *répartiteur (dispatcher)* de transactions qui décide quoi faire avec chaque transaction
- Cela distribue un appel de procédure ou un message à l'une d'une série de composantes qui va *traiter* la transaction

Exemple d'un système transactionnel



L'architecture courtier et les principes de design

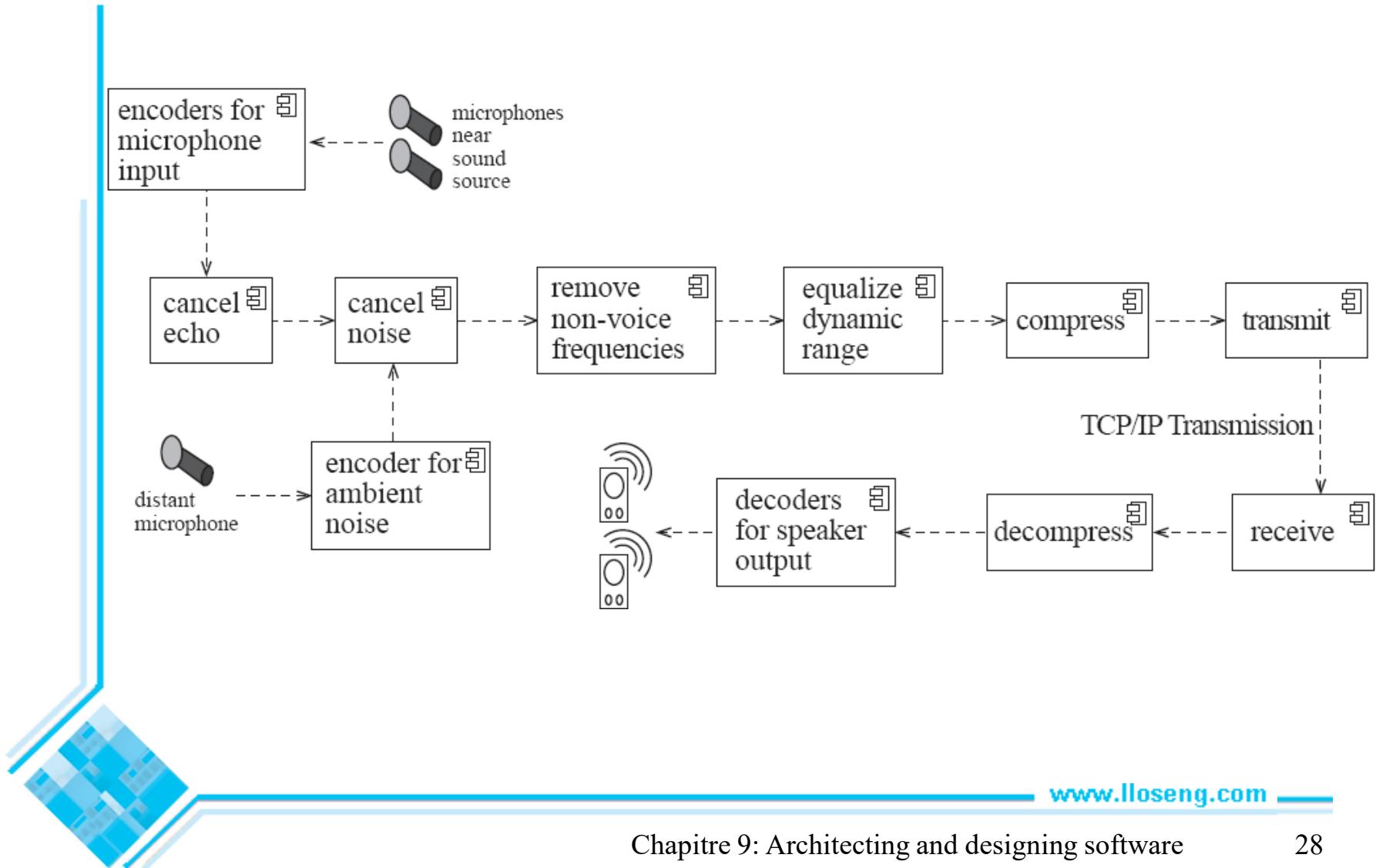
- 
1. *Diviser pour régner*: les différents processus de traitement sont en fait des divisions naturelles du système.
 2. *Accroître la cohésion*: les différentes composantes de traitement sont naturellement cohésifs.
 3. *Réduire le couplage*: la séparation du répartiteur et les composantes de traitement tend à réduire le couplage.
 7. *Accroître la flexibilité*: il est très facile d'ajouter de nouvelles transactions.
 11. *Concevoir de façon défensive*: il est possible d'ajouter des assertions dans chacune des composantes de traitement.

L'architecture en filtres

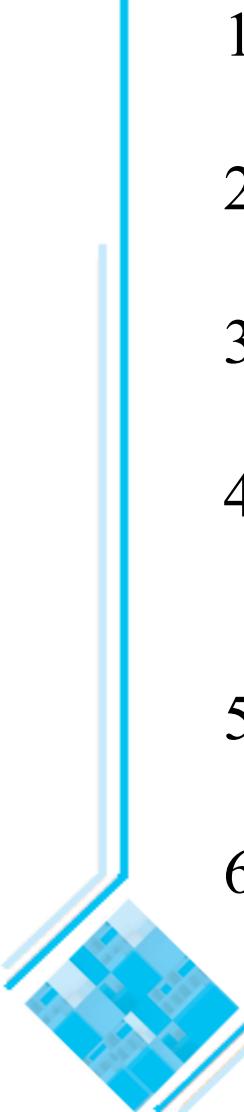
Un flot de données, souvent relativement simples, passe d'un processus à un autre pour être transformées

- Les données sont donc transformées par les traitements subits.
- Les différents processus travaillent en concurrence.
- Cette architecture est très flexible.
 - Presque toutes les composantes peuvent être retirées.
 - Les composantes peuvent être remplacées.
 - De nouvelles composantes peuvent être insérées.
 - Certaines composantes peuvent être réordonnées.

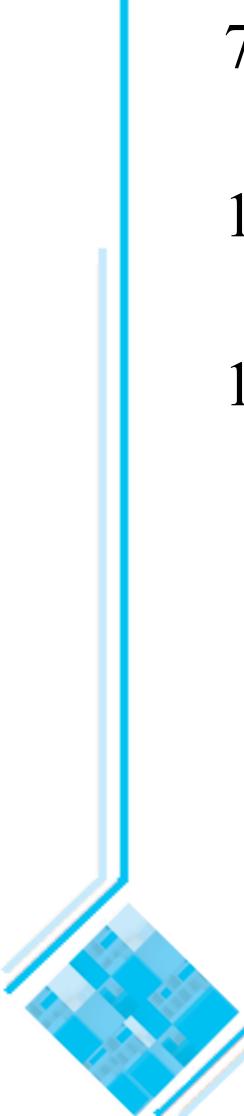
Exemple d'architecture en filtres



L'architecture en filtres et les principes de design

- 
1. *Diviser pour régner*: les différents filtres peuvent être conçus indépendamment.
 2. *Accroître la cohésion*: ces différents processus ont une cohésion fonctionnelle.
 3. *Réduire le couplage*: les différents filtres ont une seule entrée et une seule sortie.
 4. *Accroître l'abstraction*: chacun des filtres, dont les détails internes sont masqués, constitue une bonne abstraction.
 5. *Accroître la réutilisabilité*: les différents filtres peuvent être réutilisé dans différents contextes.
 6. *Accroître la réutilisation*: il est souvent possible de trouver des filtres à réutiliser.

L'architecture en filtre et les principes de design

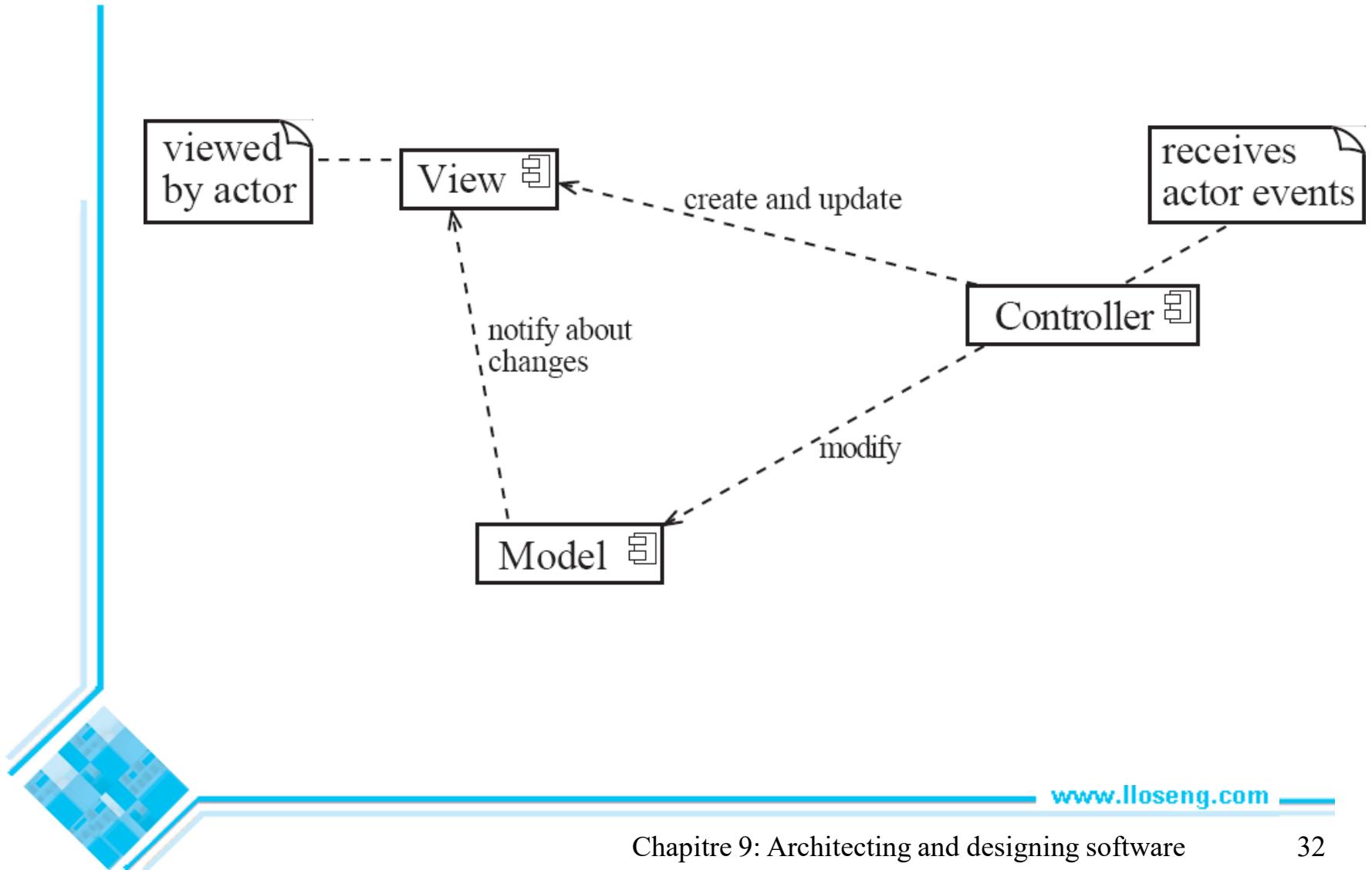
- 
- 7. *Accroître la flexibilité*: la séquence de filtres peut être modifiée à volonté.
 - 10. *Faciliter les tests*: les différents filtres sont aisément testables individuellement.
 - 11. *Concevoir de façon défensive*: les entrées du filtre peuvent être rigoureusement vérifiées.

L'architecture Modèle-Vue-Contrôleur (MVC)

Ce patron architectural est utilisé pour séparer l'interface utilisateur des autres parties du système

- Le *modèle* contient les classes dont les instances seront visualisée et manipulée
- La *vue* contient les objets utilisés pour afficher les données du modèle
- Le *contrôleur* contient les objets qui gère les interactions de l'utilisateur avec la vue et le modèle
- Le patron de conception Observateur est normalement utilisé pour séparer la vue du modèle

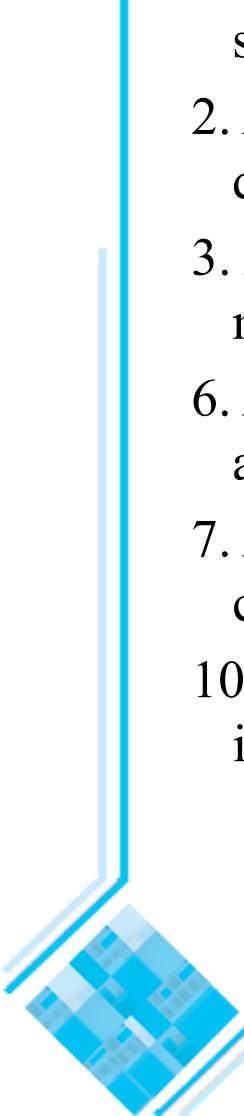
Exemple d'architecture MVC pour IU



Exemple d'architecture MVC pour le Web

- La composante *Vue* génère le code HTML à afficher par le navigateur.
- Le *Contrôleur* est la composante qui interprète les requêtes «HTTP» provenant du navigateur.
- Le *Modèle* est la composante qui gère les informations.

L'architecture MVC et les principes de design

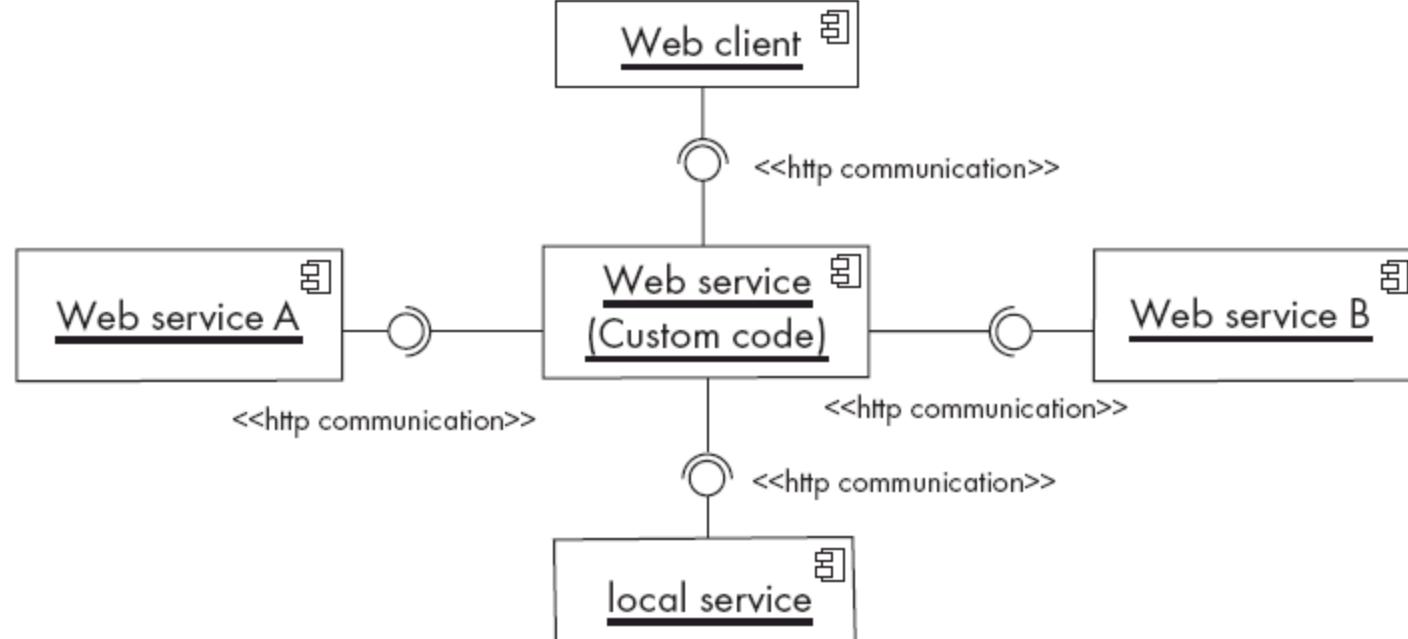
- 
1. *Diviser pour régner*: les trois composantes peuvent être conçues séparément.
 2. *Accroître la cohésion*: les composantes ont une cohésion plus forte que si ces composantes étaient combinées.
 3. *Réduire le couplage*: la communication entre les composantes est minimale.
 6. *Accroître la réutilisation*: la vue et le contrôleur se construisent avec des éléments réutilisables.
 7. *Accroître la flexibilité*: il est habituellement aisément d'apporter des changements aux différentes composantes.
 10. *Faciliter les tests*: il est possible de tester l'application indépendamment de l'IU.

L'architecture orienté Service

Cette architecture organise une application en une collection de services communiquant entre eux via des interfaces bien définies

- Sous Internet, ce sont des services Web
- Un service web est une application accessible via l'Internet, pouvant être intégrée à d'autres services afin de former un système complet
- Les différentes composantes communiquent généralement entre elles en utilisant le standard XML (ou JSON)

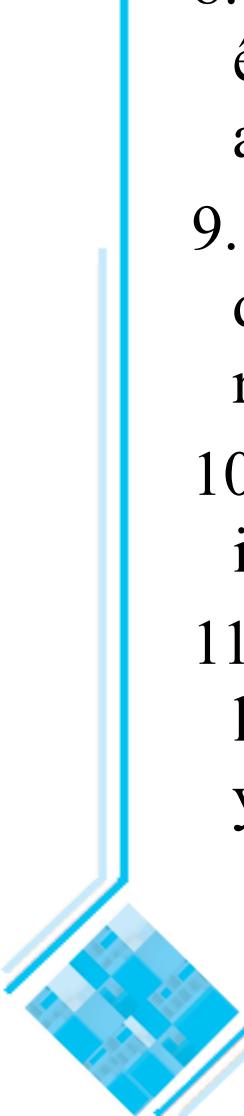
Exemple d'une application orientée service



L'architecture orientée service et les principes de design

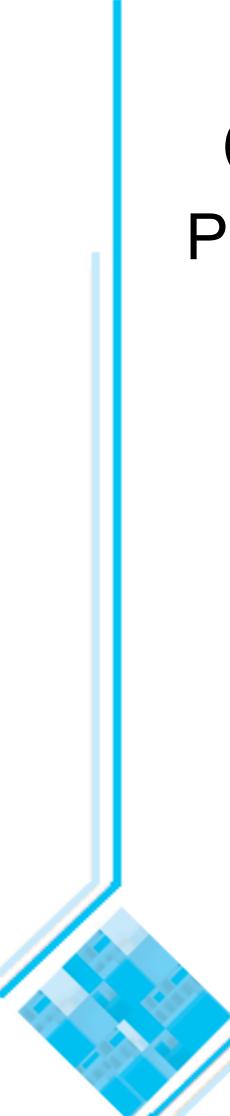
- 1. *Diviser pour régner*: l'application est construite à partir de services indépendants.
- 2. *Accroître la cohésion*: les services Web sont structurés en couches et ont une excellente cohésion fonctionnelle.
- 3. *Réduire le couplage*: les applications Web sont très peu couplées puisque bâties à l'aide de composantes distribuées.
- 4. *Accroître la réutilisabilité*: un service Web est une composante hautement réutilisable.
- 5. *Accroître la réutilisation*: une application Web est conçue à partir de services Web existants.
- 6. *Accroître la flexibilité*: il est habituellement facile d'apporter des changements aux différentes composantes.

L'architecture orientée service et les principes de design

- 
8. *Anticiper l'obsolescence*: les services obsolètes peuvent être remplacés par de plus récents sans affecter les applications l'utilisant.
 9. *Concevoir des designs portables*: un service peut être déployé sur toute plateforme supportant les standards requis.
 10. *Faciliter les tests*: chaque service peut être testé indépendamment.
 11. *Concevoir de façon défensive*: un service Web impose le design défensif puisque plusieurs applications peuvent y accéder.

Sommaire des architectures vs les principes de design

	1	2	3	4	5	6	7	8	9	10	11
Multi-layers											
Client-server											
Broker											
Transaction processing											
Pipe-and-filter											
MVC											
Service-oriented											



Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 10:

Testing to Ensure High Quality

Définitions de base

- **Panne:** Un comportement inacceptable ou une exécution incorrecte du système
 - La fréquence des pannes mesure la fiabilité du système
 - L'objectif est d'atteindre un taux d'échec très faible.
 - Une panne résulte de la violation d'une exigence explicite ou implicite
- **Faute/Défaut/Bug:** Une faille dans un aspect du système qui contribue ou peut contribuer à l'apparition d'une ou plusieurs **pannes**.
 - Peut se trouver dans les exigences, le design ou le code
 - Il peut prendre plusieurs défauts pour provoquer une **panne** particulière
 - Aussi appelé **bug**
- **Erreur:** Dans le contexte du testing, une décision inappropriée prise par un développeur qui cause l'introduction d'un **défaut**

Types de tests

Tests en boîte blanche (white-box testing):

- **Couverture d'instructions**- Cette technique vise à exercer toutes les instructions de programmation avec des tests minimaux.
- **Couverture des branches** - Cette technique exécute une série de tests pour garantir que toutes les branches sont testées au moins une fois.
- **Couverture de chemins** - Cette technique correspond au test de tous les chemins possibles, ce qui signifie que chaque instruction et branche est couverte.

Test en boîte noire (black-box testing):

- Classes d'équivalence + tests de limites

Tests en boîte blanche (white-box)

Les testeurs ont accès au code et au design du système

- Ils peuvent
 - exécuter un débogueur
 - générer une trace
 - injecter un nouveau code pour voir ce qui se passe

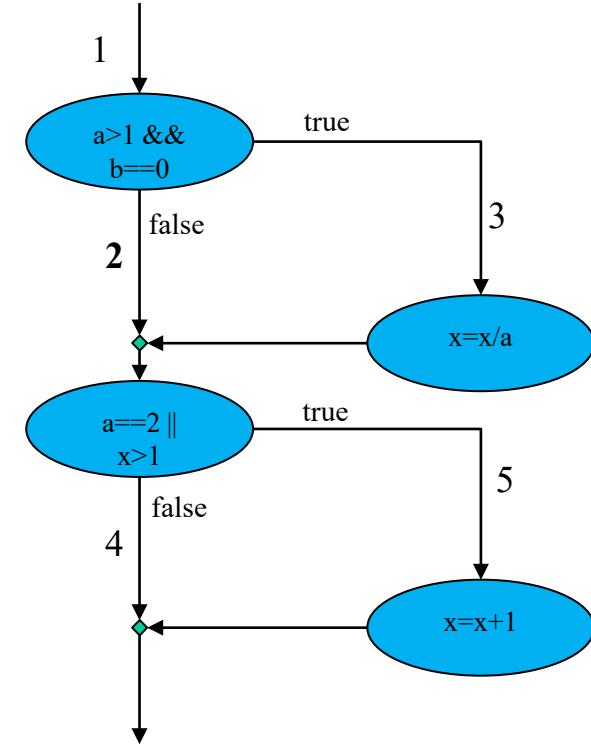
Flot de données pour tests en boîte blanche

Voir la diapositive suivante pour un exemple de flot de données

- Chaque instruction ou décision (exemple: *if* ou *while*) correspond à un *nœud* dans le graphe
- La stratégie de test doit atteindre une couverture des instructions et des branches; l'objectif peut être de:
 - couvrir tous les **nœuds** possibles (plus simple) - **couverture d'instructions**
 - couvrir tous les **arêtes** possibles (le plus efficace) - **couverture de branches**
 - couvrir tous les **chemins** possibles (souvent irréalisables) - **couverture de chemins**

Exemple: flot de données pour tests en boîte blanche

```
int proc(int a, int b, int x) {  
    if ((a>1) && (b==0))  
        x = x/a;  
    if ((a==2)|| (x>1))  
        x = x+1;  
    return x;  
}
```

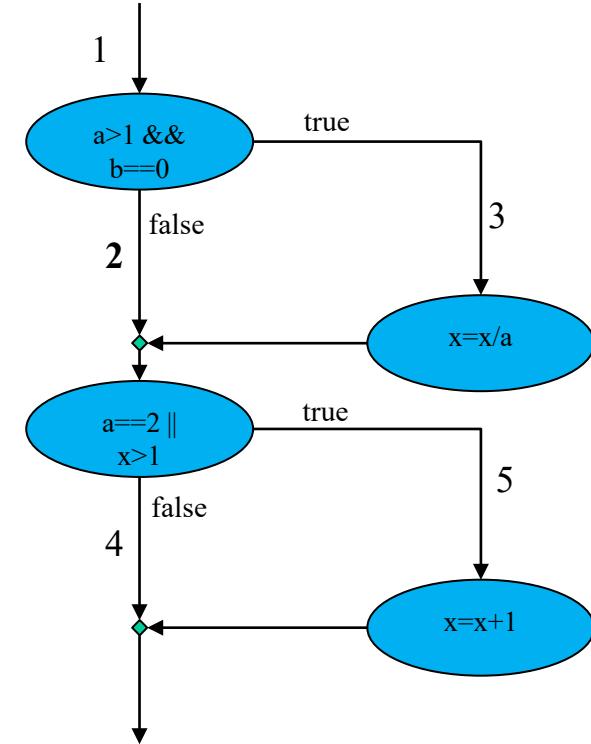


Exemple – Couverture d'instructions

```
int proc(int a, int b, int x) {  
    if ((a>1) && (b==0))  
        x = x/a;  
    if ((a==2)|| (x>1))  
        x = x+1;  
    return x;  
}
```

Couverture d'instructions:

- cas de test pour 1 – 3 – 5
- exemple: a = 2, b = 0, x = 3

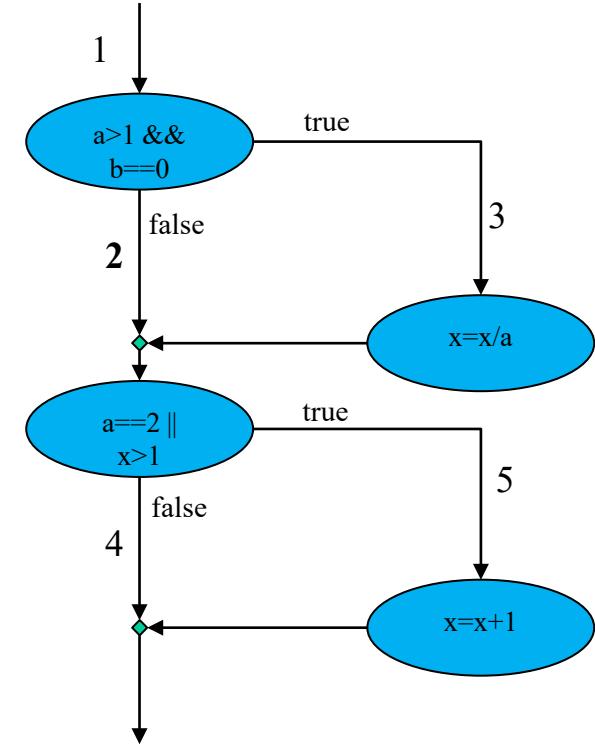


Exemple – Couverture de branches

```
int proc(int a, int b, int x) {  
    if ((a>1) && (b==0))  
        x = x/a;  
    if ((a==2)|| (x>1))  
        x = x+1;  
    return x;  
}
```

Couverture de Branches:

- cas de test pour 1 – 2 – 5 (a = 2, b = 2, x = -1)
- cas de test pour 1 – 3 – 4 (a = 3, b = 0, x = 1)

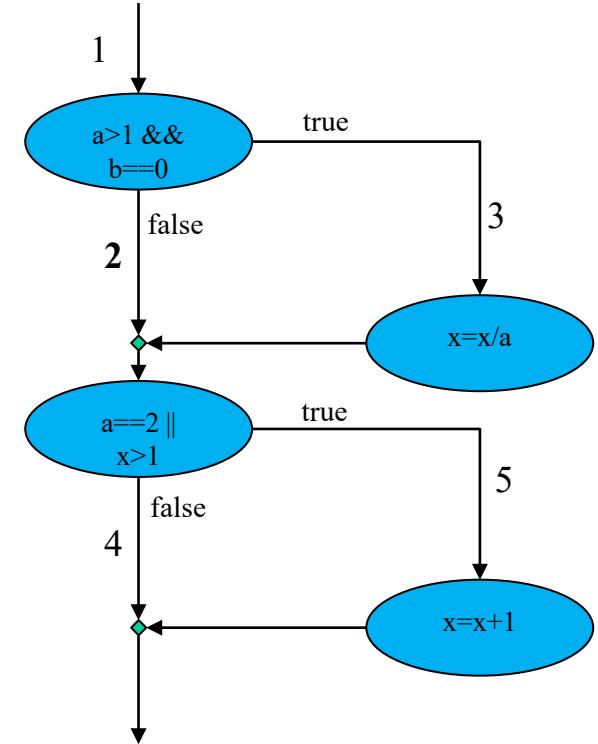


Exemple – Couverture de chemins

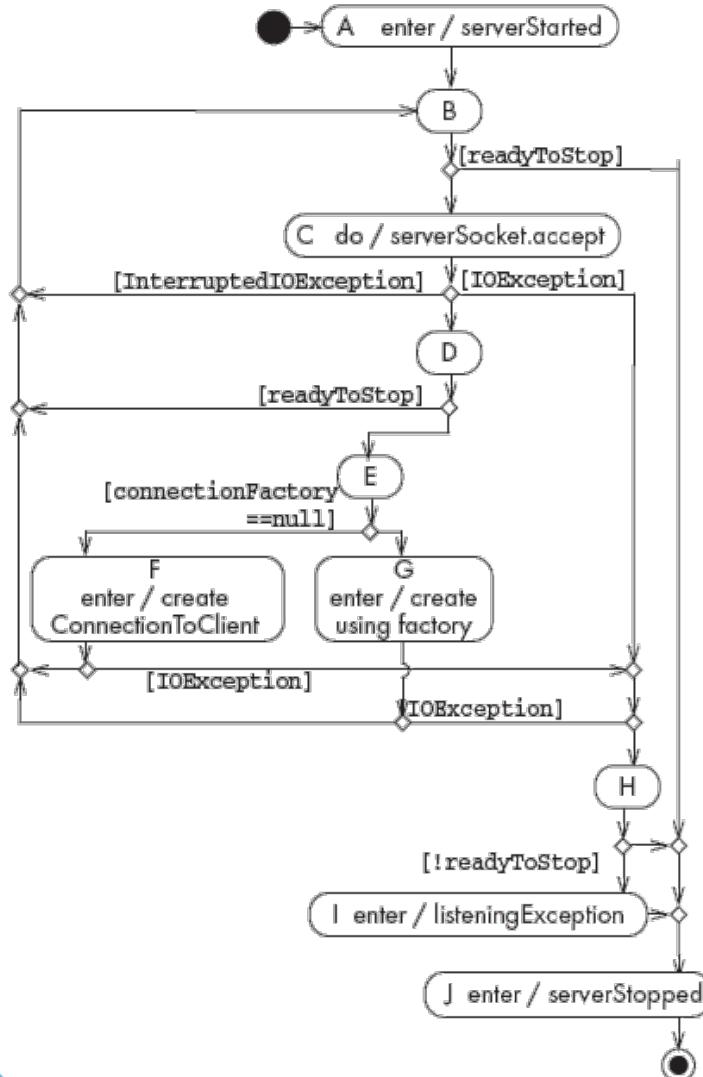
```
int proc(int a, int b, int x) {  
    if ((a>1) && (b==0))  
        x = x/a;  
    if ((a==2)|| (x>1))  
        x = x+1;  
    return x;  
}
```

Couverture de chemins:

- Cas de test pour 1-2-4 ($a=3, b=2, x=0$)
- Cas de test pour 1-3-4 ($a=3, b=0, x=1$)
- Cas de test pour 1-2-5 ($a=2, b=2, x=-1$)
- Cas de test pour 1-3-5 ($a=3, b=0, x=9$)



Flot de données pour tests en boîte blanche



```
readyToStop= false;  
serverStarted(); // A  
try  
{  
    while(!readyToStop) // B  
    {  
        try  
        {  
            Socket clientSocket = serverSocket.accept(); // C  
            synchronized(this)  
            {  
                if (!readyToStop) // D  
                {  
                    if (connectionFactory == null) { // E // F  
                        new ConnectionToClient( // F  
                            this.clientThreadGroup, clientSocket, this);  
                    } else {  
                        connectionFactory.createConnection( // G // G  
                            this.clientThreadGroup, clientSocket, this);  
                    }  
                }  
            }  
        } catch (InterruptedException exception) {}  
    }  
    catch (IOException exception)  
    {  
        if (!readyToStop) // H  
        {  
            listeningException(exception); // I // I  
        }  
    }  
    finally  
    {  
        readyToStop = true; // J  
        connectionListener = null;  
        serverStopped();  
    }  
}
```

Tests en boîte noire

Les testeurs fournissent au système des entrées et observent les sorties

- Ils ne peuvent pas voir:
 - Le code source
 - Les données internes
 - Tout document de conception décrivant les composants internes du système

Les classes d'équivalences

- Il est généralement **impossible** de tester *chaque valeur possible* comme entrée
 - E.g. Chaque entier
- Au contraire, on divise les entrées possibles dans de groupes que vous croyez qui seront traités de manière similaire par tous les algorithmes.
 - Ces groupes sont appelés **classes d'équivalence**
 - Un testeur doit créer seulement 1-3 tests par classe d'équivalence

Exemples de classes d'équivalence

- Entrée valide pour le mois (1-12)
 - Les CE sont: [-∞..0], [1..12], [13.. ∞]
- Entrée valide pour une chaîne de caractères représentant 10 marques différentes de voiture (Mazda, Nissan,...etc)
 - Les CE sont
 - 10 classes, une pour chaque marque
 - Une classe représentant toutes les autres chaînes

Combinations de classe d'équivalence

- **Explosion combinatoire**: vous ne pouvez pas tester chaque combination possible des classes d'équivalence
 - S'il y a 4 entrées avec 5 valeurs possibles, il y aura 5^4 (i.e. 625) combinations possibles des classes d'équivalence
- Au contraire, on teste:
 - Chaque** classe d'équivalence
 - On **combine** seulement lorsqu'une entrée pourrait affecter une autre
 - On choisit d'autres combinaisons **au hasard**

Exemple – Combination de classes d'équivalence

- Une entrée valide est soit 'Metric' ou 'US/Imperial'
 - Les CE sont:
 - Metric, US/Imperial, Autre
- Une autre entrée valide est vitesse maximale: 1 à 750 km/h ou 1 to 500 mph
 - La validité dépend si l'on choisit Metric ou US/Imperial
 - Les CE sont:
 - $[-\infty..0]$, $[1..500]$, $[501..750]$, $[751..\infty]$
- Quelque combinaison des tests
 - Metric, $[1..500]$ valide
 - US/Imperial, $[1..500]$ valide
 - Metric, $[501..750]$ valide
 - US/Imperial, $[501..750]$ invalide

Analyse des Valeurs Aux Bornes

- Plus d'erreurs logicielles se produisent aux bornes des classes d'équivalence
- Sélectionner les éléments juste à et autour des bornes de chacunes des CEs
 - E.g. Le numéro 0 cause souvent des problèmes
- *Ex:* Si l'entrée valide est un mois (1-12)
 - Testez 0, 1, 12 et 13
 - Testez avec un gros chiffre positif et un très petit chiffre négatif

Développement centré sur les Tests (Test-driven development)

Pratique courante:

- Écrire tous les tests avant d'écrire le code qui effectue les opérations (i.e. **test-first**)
- Utilisez des outils comme
 - junit pour exécuter les tests
 - Ant pour automatiser l'exécutions des tests
- Utilisez les tests comme spécification du système
- Quand vous écrivez un test, assurez vous qu'il ne passe pas
 - Ensuite écrivez le code qui effectue l'opération
 - Ensuite assurez vous que le test passe
- Pour tester les UI : Selenium

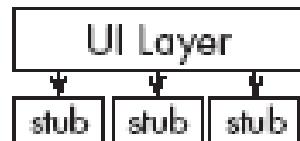
10.9 Stratégies de test de grands systèmes

Tests Big Bang et tests incrémentiel

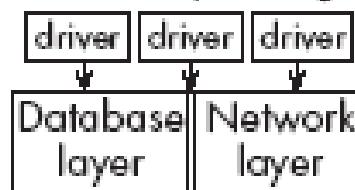
- Dans les tests Big Bang, vous prenez le système entier et le testez comme une unité
- Une meilleure stratégie dans la plupart des cas est le *test incrémentiel*:
 - Vous testez chaque sous-système individuellement
 - Continuez les tests à mesure que vous ajoutez de plus en plus de sous-systèmes au produit final
 - Les tests incrémentaux peuvent être effectués horizontalement ou verticalement, selon l'architecture
 - Les tests horizontaux peuvent être utilisés lorsque le système est divisé en sous-applications distinctes

Stratégies verticales pour les tests d'intégration incrémentiels

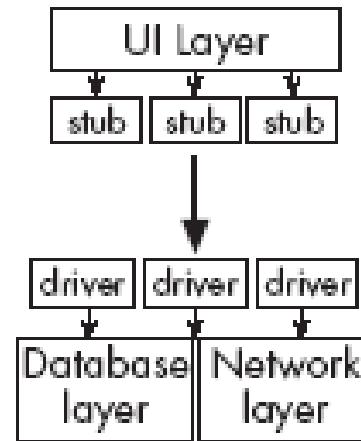
Top-down testing



Bottom-up testing



Sandwich testing



Fully
integrated
system

Les tests de régression

Lors du test manuel

- Tester un sous-ensemble bien choisi des cas de test précédemment réussis

Lors du test automatique

- Exécutez tous les tests, car il est généralement rapide

La «loi de conservation des bugs»:

- *Le nombre de « bugs » restants dans un grand système est proportionnel au nombre de « bugs » déjà corrigés*

10.13 Difficultés et risques liés à l'Assurance Qualité (AQ)

Il est très facile d'oublier de tester certains aspects d'un système logiciel:

- «*Exécuter le code plusieurs fois*» ne suffit pas.
- *Oublier certains types de tests diminue la qualité du système.*

Il existe un conflit entre l'atteinte de niveaux de qualité adéquats et le fait qu'il faut livrer le système aussitôt possible

- *Créez un département distinct pour superviser l'AQ.*
- *Publiez des statistiques sur la qualité.*
- *Prévoyez suffisamment de temps pour toutes les activités.*

Difficultés et risques liés à l'Assurance Qualité (AQ)

Les employés ont des capacités et des connaissances différentes en matière de qualité

- *Donnez aux employés des tâches qui correspondent à leur personnalité naturelle.*
- *Former les employés aux techniques de test et d'inspection.*
- *Donner aux employés du feedback concernant leurs performances en termes de production de logiciels de qualité.*
- *Faites travailler les développeurs et les mainteneurs pendant plusieurs mois dans une équipe de test.*