POO

- Constitution des modules
- Ecrire des modules
- Principes POO

Principe d'encapsulation

- Cacher le fonctionnement interne d'un module
- Les changements internes au module n'ont pas d'influence sur le code extérieur
- On cherche à obtenir des interfaces (au sens de la chimie)
- Concepts importants : contrat et responsabilité

Interface

- Quand deux matériaux ont une interface, cela veut dire qu'il n'y a pas d'échange de matière entre les deux
  - exemple : l'huile et l'eau
  - contre-exemple : l'or et l'argent
- On peut les séparer

Interface

- On cherche à écrire des modules qui ont des interfaces (au sens chimique)
- On veut que ni les corrections, ni les bugs ne se propagent d'un module à l'autre
- En particulier, les changements d'implémentation d'un module ne doivent pas avoir d'incidence sur les modules qui l'utilisent

Module client

- Quand deux modules interagissent, celui qui **utilise l'autre** est appelé module client
  - Ex Voiture/Conducteur
- Le conducteur utilise l'interface de la voiture et pas l'inverse
- Ex Administrateur/Administré
- Les deux modules font des demandes croisées. Les deux modules sont clients l'un de l'autre.

Interface

- L'interface est la partie publique d'un module
- Ce doit être la partie stable d'un module
- Un module client utilise l'interface
- Si l'interface change, les modules clients doivent changer

Interface

- **Une bonne partie du travail de conception est la définition des interfaces entre les différents modules**
- Il est important de bien définir l'interface
  - Syntaxique : prototype
  - Sémantique :
    - fonctionnel
    - contrat complet (prérequis, cas d'erreur, effet de bord?, ...)

Implémentation

- Un module est aussi composé de l'implémentation
- L'implémentation réalise les fonctionnalités de l'interface

De l'art d'écrire des Modules

- Le travail d'étude se différencie du travail de production
  - il s'applique à quelque chose de Neuf.
  - Ainsi le risque d'erreur y est beaucoup plus prégnant.
    Toutes les méthodes cherche a réduire le coût de la correction d'erreur avec une stratégie qui consiste a faire cette détection au plus tôt.

- L'art d'écrire les module à pour objectif de réduire le nombre d'erreurs.

Construction des modules

- Principe d'ouverture/fermeture
- Interfaces
- Fermeture commune
- Stabilité et abstraction
- Dépendances acycliques
- Livraison
- Ré-ouvrir ?

Principe d'ouverture/fermeture

- On cherche à avoir des modules ouverts/fermés
- Cette stratégie s'applique quelle que soit le type de module (classe, paquetage, .c)

Interfaces

- Le choix d'une interface est une activité primordiale qui doit être identifiée et maîtrisée en tant que telle
- C'est un savoir faire spécifique
- La qualité des interfaces est un préalable à la qualité du logiciel

Interfaces

- Objectifs pour une interface
  - réduire les risques d'ouverture
  - assurer la compréhensibilité
  - assurer la réutilisabilité
  - continuité

Petites interfaces

- Les interfaces pléthoriques vont à l'encontre de la com-préhensibilité du module
- Une interface importante n'aide pas à la réutilisabilité du module
- Si l'interface est grande, le module et la maintenance sont complexes

Peu d'interfaces

- Il faut réduire le nombre d'interfaces utilisées par un module :
  - un module doit être le client d'un minimum d'autres
- + de fournisseurs => + de chances d'avoir à changer
- + de fournisseurs => + de complexité

Interfaces explicites

- Les modules doivent avoir une interface explicite où rien n'est caché ou tordu
- La documentation du module devrait pouvoir se limiter à l'interface

Fermeture commune

- Les entités qui dépendent d'un même concept doivent être placé dans le même module
- Si le concept change, les classes concernées sont toutes dans le même paquetage
- **L'unité de fermeture est le module**
- Changer un concept force la réouverture, il est important qu'un nombre minimal de modules soit rouverts

Stabilité

- La stabilité est l'inverse du risque de changement
- Plus le module est client d'autre modules, moins il est stable
- Mesure mathématique : hauteur dans le graphe de dépendance

Valeur de la stabilité

- Plus un module est fermé donc stable, plus il a de valeur pour les développeurs
- Plus les modules stables (qui ont donc beaucoup de clients directs et indirects) sont effectivement stable, plus l'application est stable dans son ensemble

Stabilité et abstraction

- Plus un module est abstrait, plus il est facile de le garder stable
- Pour fabriquer un paquetage stable, il faut identifier les bonnes abstractions
  - ne rien sous-entendre sur l'implémentation
  - *plus c'est abstrait, plus c'est stable*

Stabilité

- Identifier et concevoir les paquetages qui doivent être stables est une tâche importante
- Penser que l'on peut agrandir son paquetage avec de nouvelles interfaces sans trop de danger

- *Un package ne doit dépendre que de packages plus stables que lui.*

Dépendances acycliques

- Un module doit toujours dépendre de modules plus stables que lui
- En respectant cette règle, **le graphe de dépendances ne doit pas être cyclique**
- En cas de cycle, il faut faire de *refactoring* pour éliminer le cycle

Livraison

- Un module livré est un paquet fourni aux clients
- Livraison = publication = mise-en-forme
- livré : numéro de version
  release.version.revision
- Seulement une fois livré, c'est réutilisable

Livraison / Configuration

- Penser à spécifier dans votre paquetage livré la configuration cible
  - version de logiciels et bibliothèques utilisés
  - configuration hardware
  - Les contraintes de configuration

Livraison => Réutilisation

- **On ne peut réutiliser qu'un paquetage livré**
- On réutilise un paquetage complet et non une classe
- Le code est la propriété de l'auteur, c'est lui qui corrige les bugs
- *« Je ne veux pas le savoir du moment que ça marche, je ne cherche pas à savoir pourquoi »*
  - … sauf au coin café, après avoir épuisé les sujets de discussions usuels

Cycle de développement

- Un cycle est
  - analyse
    - conception
      - Ecriture du Code
    - test (unitaires,...)
  - Livraison
- Plus un problème arrive tard, plus le changement est important
- Il faut des cycles courts
  - Mais ...

ne veut pas forcément dire graver le CD !

Comment ré-ouvrir ?

- Deux approches pour les modules O&F
- Interface (blackbox)
  - On ferme l'interface pas l'implémentation
- Héritage/Extension (whitebox)
  - On crée un nouveau module qui ne fait qu'étendre le module précédent

Rappel : programmation objet

- champs = état / comportement = méthodes
- Encapsulation
- Héritage
- Polymorphisme

Bonnes pratiques que vous connaissez déjà !

- Une responsabilité => un objet
- I/A/C sans abus

POO : principes de base

- Séparer ce qui varie de ce qui ne varie pas !
  - Changement plus localisés
  - Moins de risques
  - Dépendances limitées
  - Moins de risques d'ouverture

- Plus de flexibilité
- Moins de risques

POO : principes de base

- Encapsulation : la base
  - Séparer intérieur / extérieur
  - Responsabiliser
  - Contractualiser
  - → On ne se préoccupe pas de l'implémentation de ce qu'on utilise

- Encapsulation = je ne sais pas *COMMENT* tu fais
  - Je ne peux pas coder une logique qui dépend de comment ça marche !
  - Cercle vertueux de solidité

POO : principes de base

- « Program to interface, not implementation »
  - Je ne sais pas *QUI* tu es
  - Je ne peux pas coder une logique qui dépend « qui fait quoi » !
  - Cercle vertueux « d'ignorance » / de généricité

POO : principes de base

- Préférer la composition à l'héritage
  - L'héritage a été mis en avant pour la réutilisation
    - Trop !
  - Souvent, l'héritage est rigide et la composition est souple
  - Flexibilité

POO : principes de base

- OCP (Open-Closed Principles)
  - Fermeture: Le code a été écrit, testé, debuggé → on n'y touche plus
  - Ouverture: Le code est prévu pour être extensible

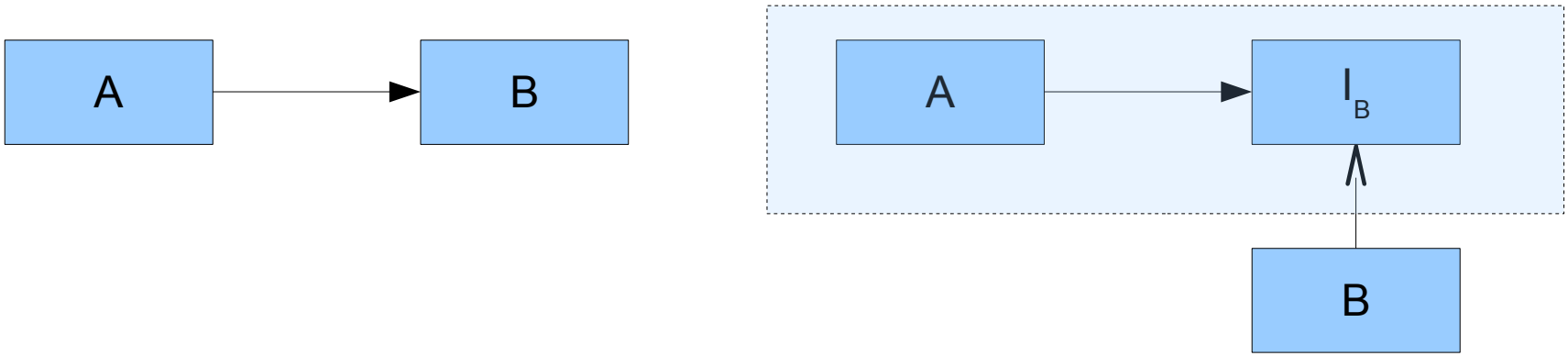
POO : principes de base

- Inversion de dépendance
  1. Réduire les dépendances sur les classes concrètes
  2. Program to interface, not implementation »
  3. → Ne dépendre QUE sur des abstractions, y compris pour les classes de haut niveau
  4. Les abstractions ne doivent pas dépendre de détails.
     → Les détails doivent dépendre d'abstractions.

  - Extensibilité

POO : principes de base

- Inversion de dépendance : comment ?



POO : principes de base

- Des règles apparemment strictes ...
  - Des interfaces cohérentes et pas trop grosses
    - Sinon, on découpe
  - Des objets qui interagissent avec peu d'objets
    - Sinon, il manque des « chefs » ou des intermédiaires
  - Ne pas parler directement aux amis de ses amis !
    - On fait transmettre les messages

Mais à appliquer avec pragmatisme !

Rudiments d'UML

UML

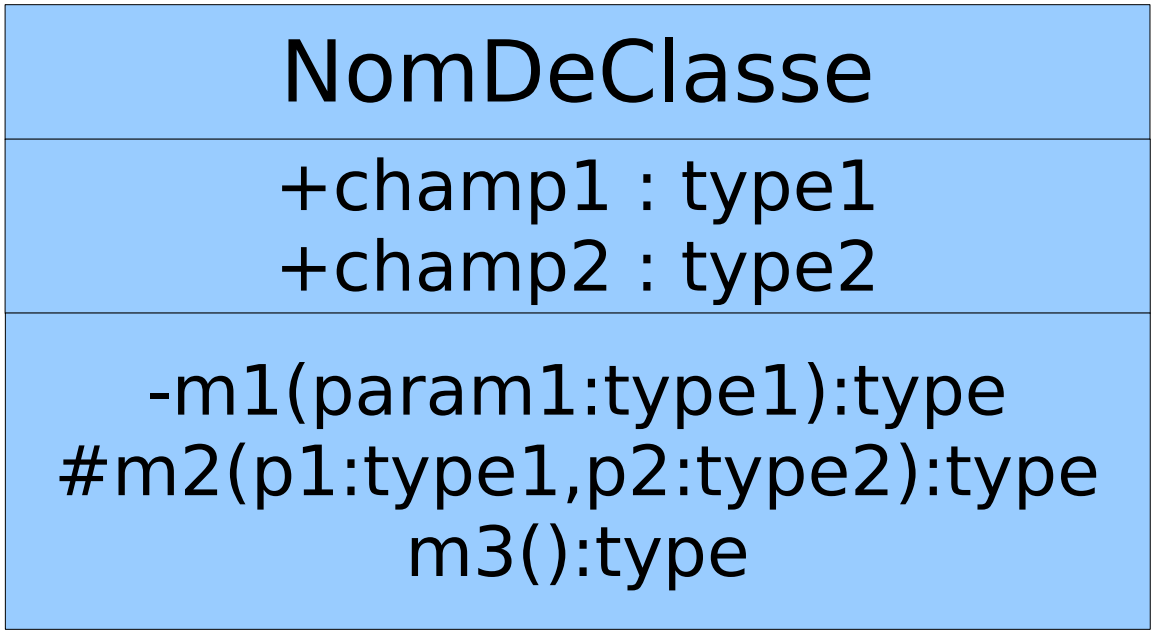
- Unified Modeling Language
- Langage uniformisé pour la spécification de modèles objets
- Graphiques standardisés
- Modèle systémique abstrait : modèle UML

Diagrammes UML

- Diagrammes structurels ou statiques
  - Diagramme de classes
  - Diagramme des paquetages
  - ...
- Diagrammes d'interaction ou dynamiques
  - Diagramme de séquence
  - ...
- Diagrammes comportementaux
  - (Diagramme des cas d'utilisation)
  - ...

Diagramme de classes

- Chaque classe est représentée par un rectangle avec
  - son nom
  - ses champs
  - ses méthodes
  - types optionnels avec notation à la pascal



+ signifie "public"
- signifie "private"
signifie "protected"
~ signifie "package"

Diagramme de classes

- Chaque interface est représentée par un rectangle surmonté d' « interface » avec
  - son nom
  - ses méthodes
  - types optionnels avec notation à la pascal

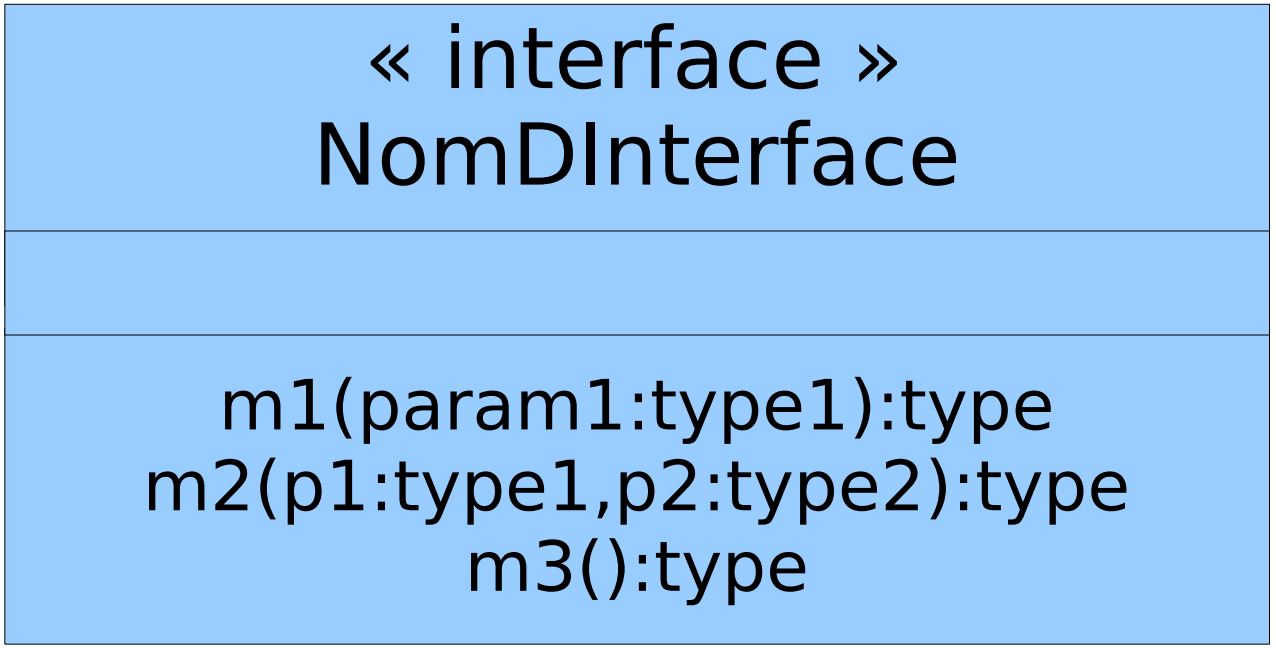
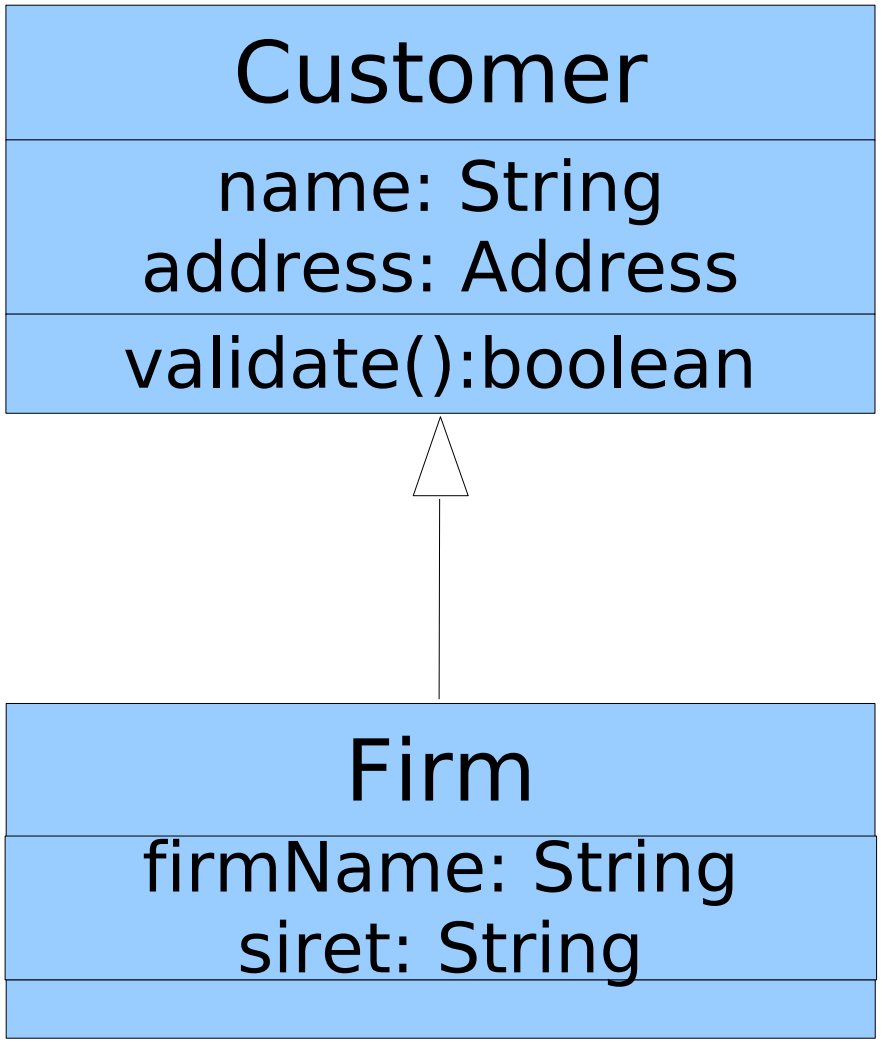


Diagramme de classes

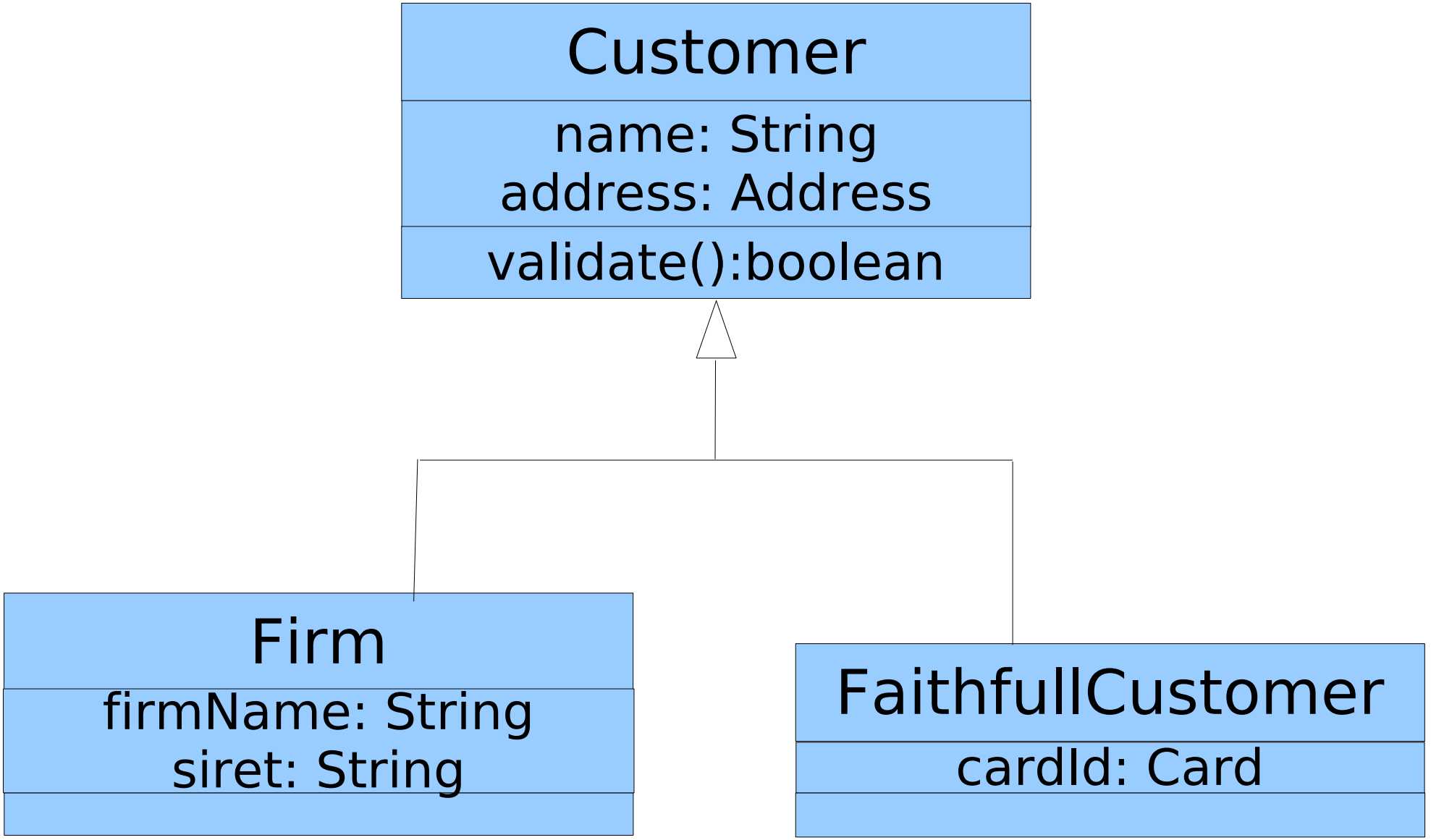
- On indique les relations entre classes et interfaces
  - généralisation (héritage)
  - réalisation (implémentation)
  - Association
    - simple ou bidirectionnelle
    - agrégation
    - composition
  - dépendance

Généralisation (héritage)

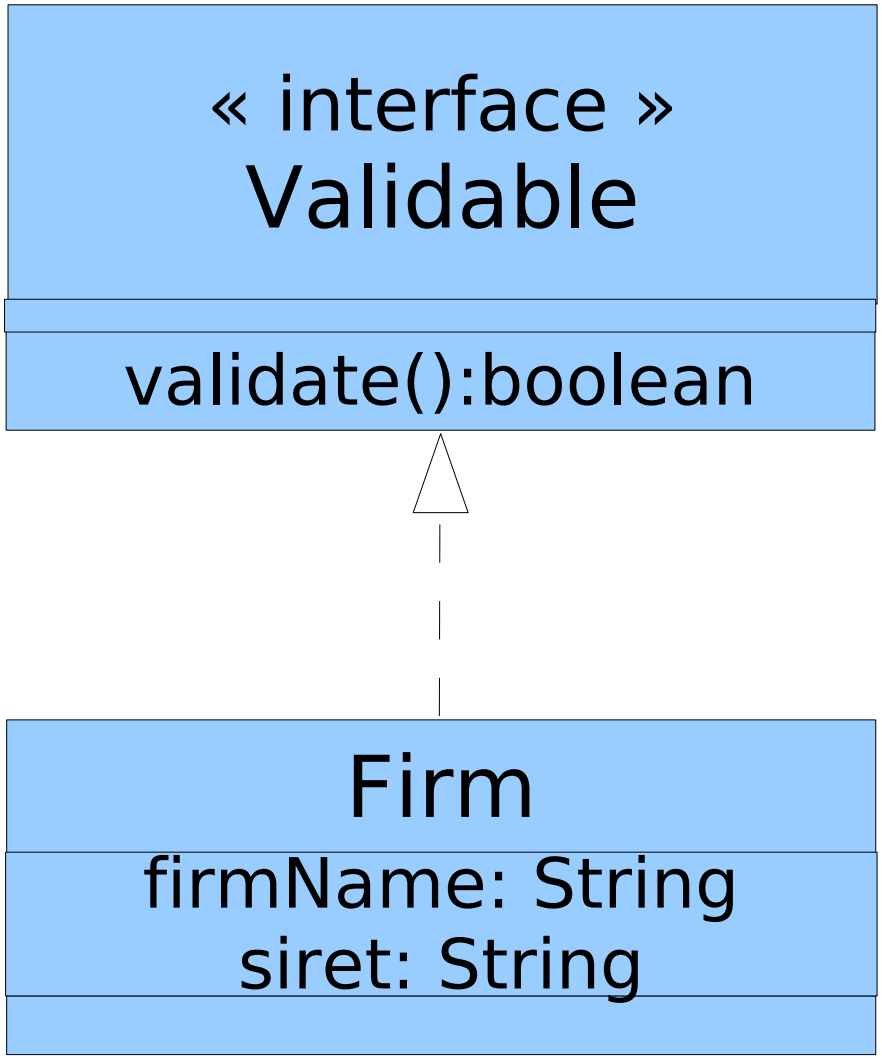
- On ne recopie pas les membres hérités



Généralisation

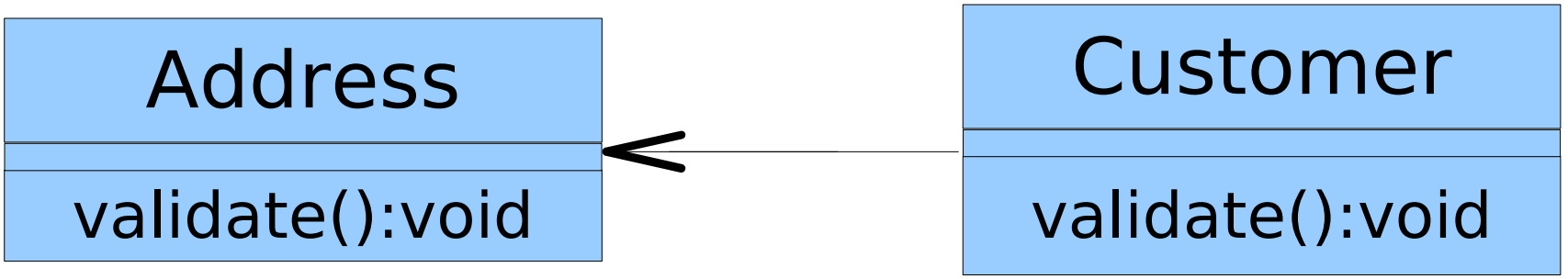


Réalisation (implémentation) :



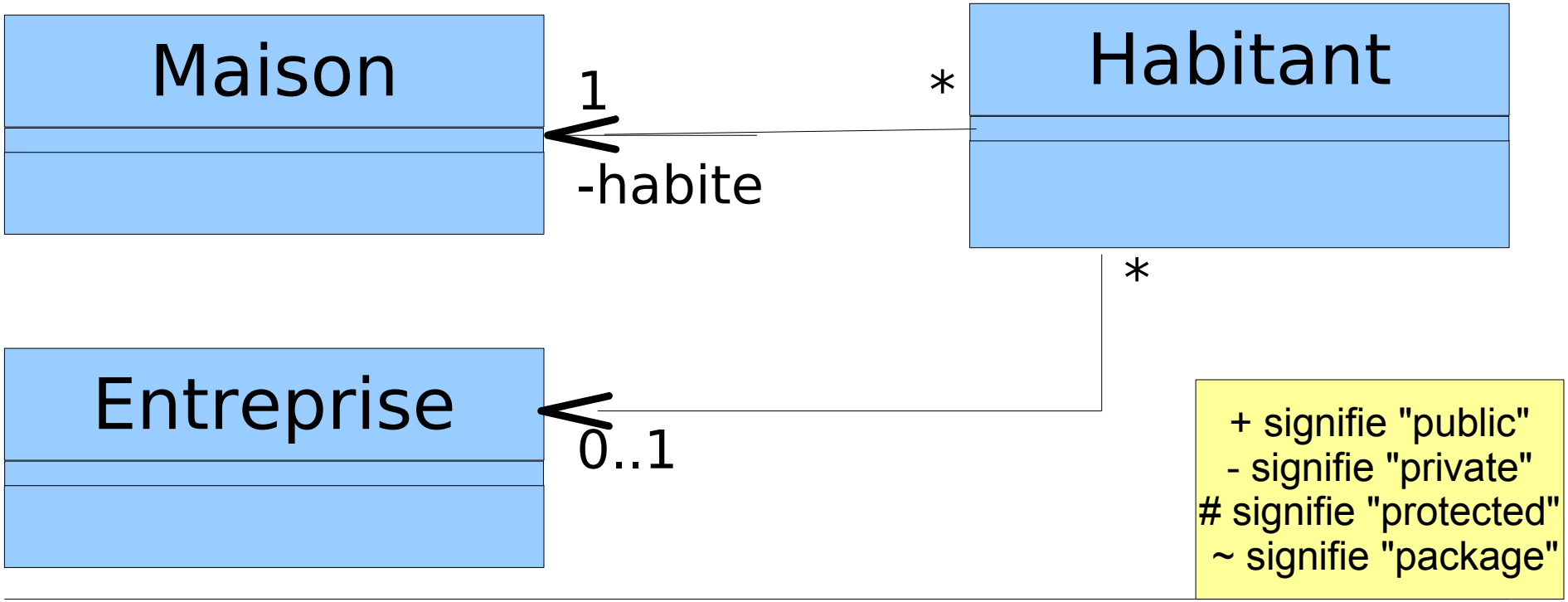
Association

- Une association signifie qu'une classe utilise une ou plusieurs instances d'une autre
- Si l'association est dans les deux sens, on ne met pas de flèche



Association

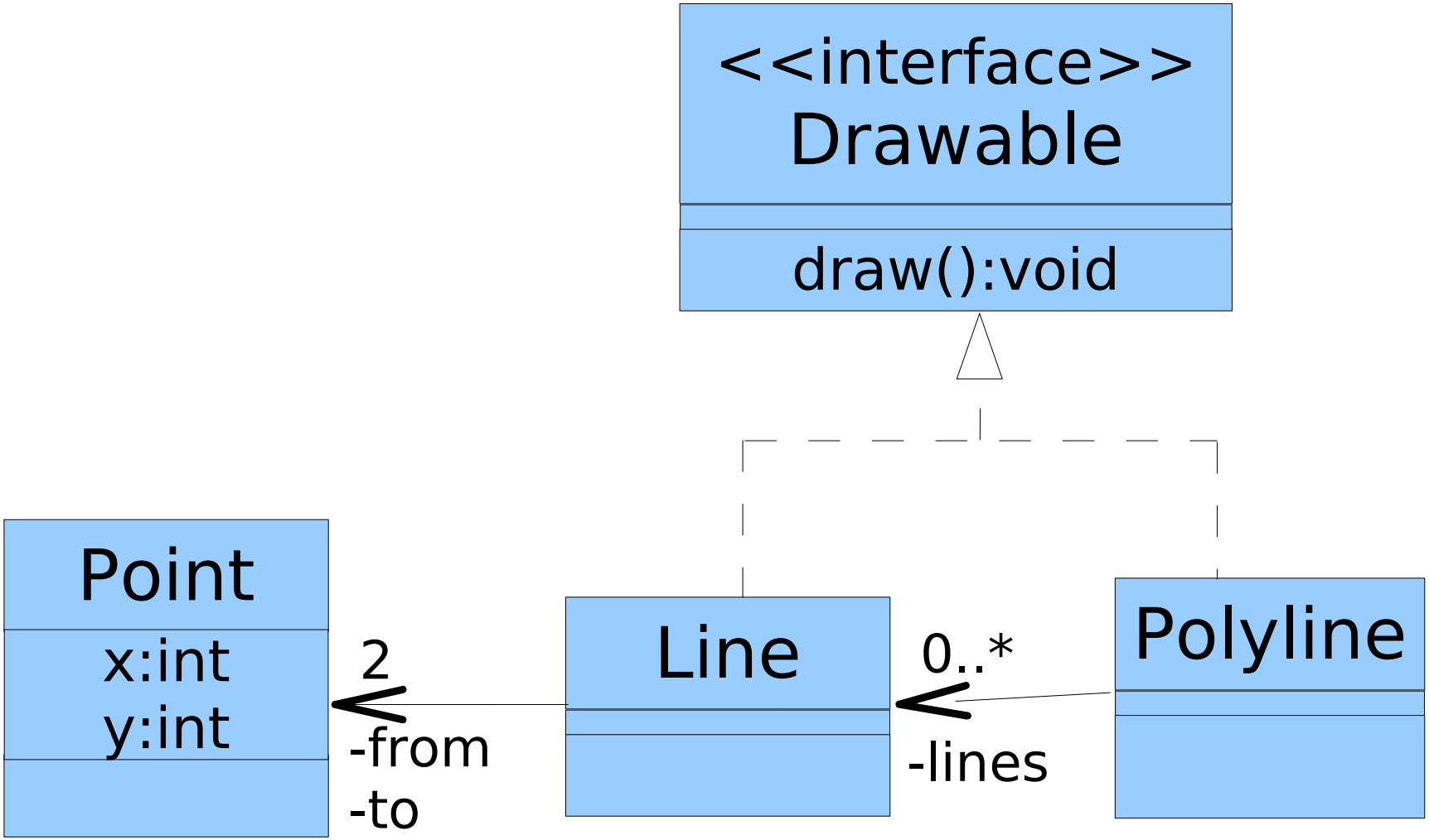
- Information complémentaire :
  - cardinalité (indiqué à l'extrémité du lien)
  - nom du champ
  - visibilité



Association

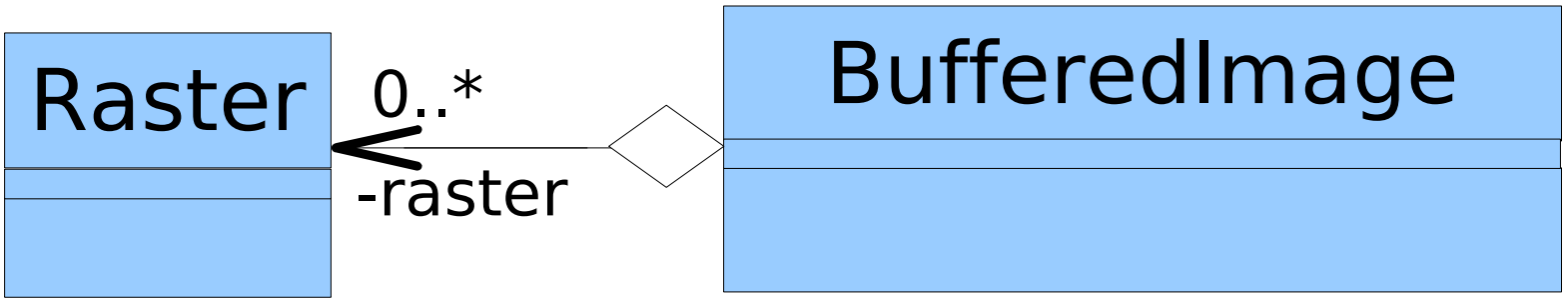
- Un cours est suivi par des étudiants (peut-être aucun) et les connaît
- Un étudiant suit des cours et les connaît

Association



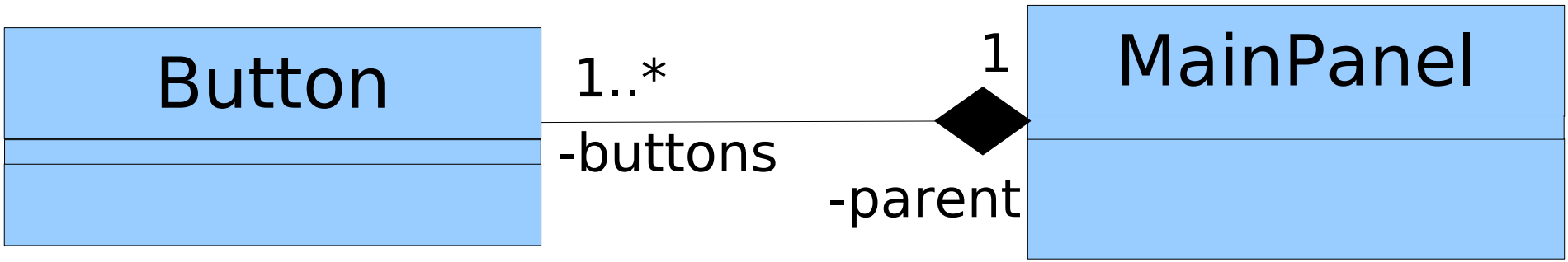
Agrégation

- L'agrégation est un cas particulier d'association :
  - Relation non symétrique, lien de subordination
  - Couplage fort
  - Relation « non vitale » les cycles de vie peuvent être indépendant



Composition

- La composition est un cas particulier d'agrégation
  - Agrégation forte
  - Cycles de vie liés
  - À un instant *t*, une instance de composant ne peut être liée qu'à un seul composé.



Composition

- En Java, si la référence sur l'instance crée quitte l'objet (getter), la composition est impossible
- En C, en théorie la composition peut être implémentée avec une structure qui contient l'autre structure (et non un pointeur)

Dépendance

- Signifie un « dépendance » de n'importe quel ordre
- On peut indiquer la nature de la dépendance

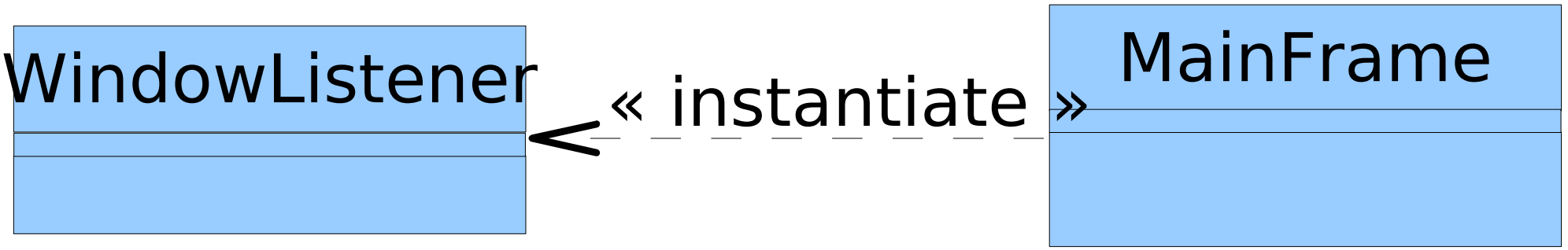


Diagramme de packages

- Simple diagramme où chaque paquetage est dans une boîte entre lesquelles on met un flèche pour dire que l'un est client de l'autre

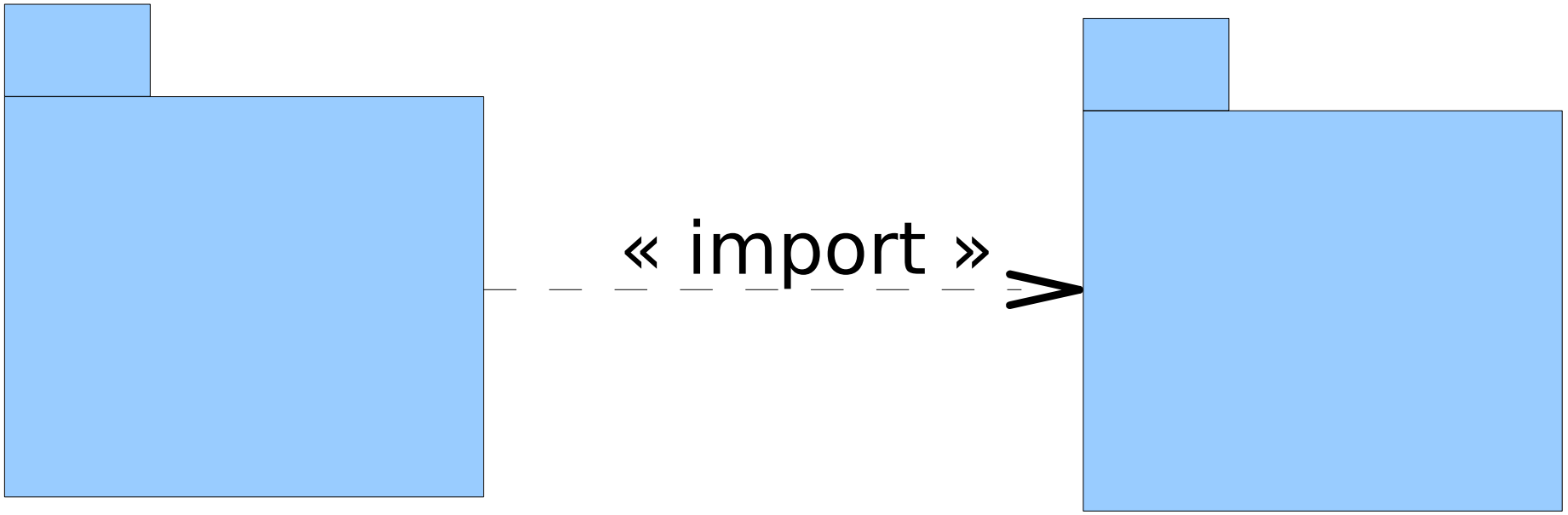


Diagramme de packages

- 2 axes
  - Cohérence
    - Nature
      - regroupement sémantique
      - finalité du service rendu
    - Evolution
      - Isoler ce qui varie
      - Niveau de stabilité
  - Indépendance
    - Faible couplage
    - Minimiser les interaction

Diagramme de séquence

- Indique ce qui se passe pendant une tâche particulière du programme
- Ce diagramme met l'accent sur les interactions entre objets
- Il permet aussi de prouver qu'une tâche cliente est possible
- En réalisant ce genre de diagrammes, on tombe souvent sur de nouvelles responsabilités.

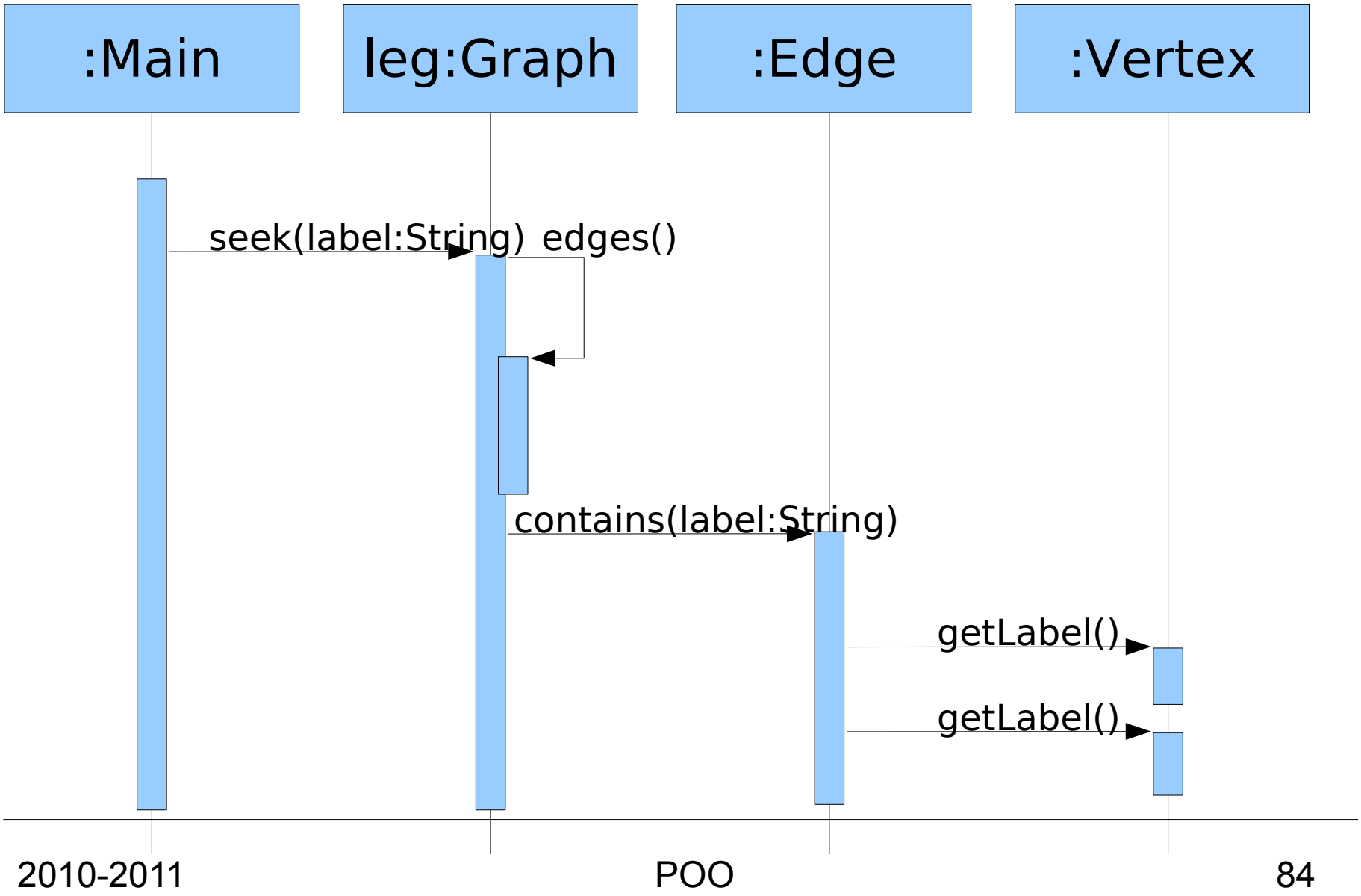



Diagramme de séquence