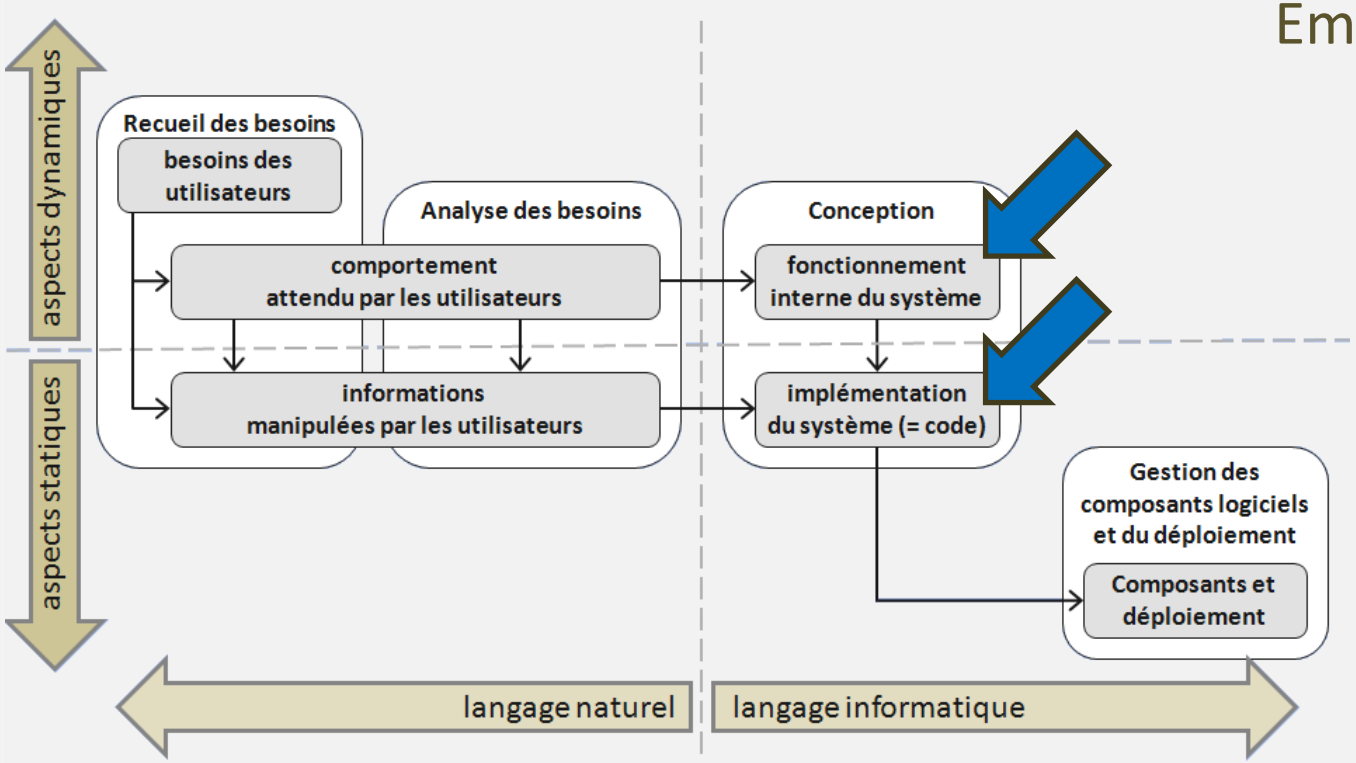
UML
Conception
Héritage

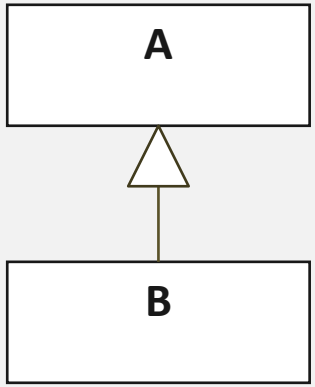
Emmanuel Pichon
2013



Héritage (generalization)

- Sens
  - Relation d'implémentation permettant la réutilisation des caractéristiques d'une classe (attributs, opérations et associations)

- Notation UML
  - Flèche en trait plein avec un triangle creux
  - UML permet l'héritage multiple



- Correspondance Java

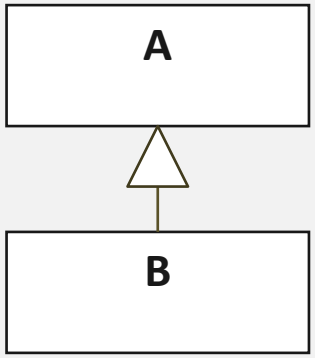
```
class B extends A {…
```

Rien à voir avec «extend» entre cas d'utilisation

  - Java ne permet pas l'héritage multiple entre classes
  - Java permet l'héritage multiple entre les interfaces
  - (Java autorise l'implémentation de multiples interfaces)

Héritage en conception

- Relation entre deux classes
  - A est la super classe (ou classe mère)
  - B est la sous classe (ou classe fille)



- La sous classe
  - Hérite de la responsabilité de la super classe et la spécialise à un contexte
  - Hérite des opérations, des attributs et des associations de la super classe
  - Peut définir des opérations, des attributs et des associations supplémentaires
  - Peut redéfinir des opérations héritées
    - Même signature (nom + paramètres) mais contenu différent (code)
    - Ne pas confondre avec la surcharge (possible aussi hors héritage)
      - Même nom mais paramètres différents (nombre ou type)

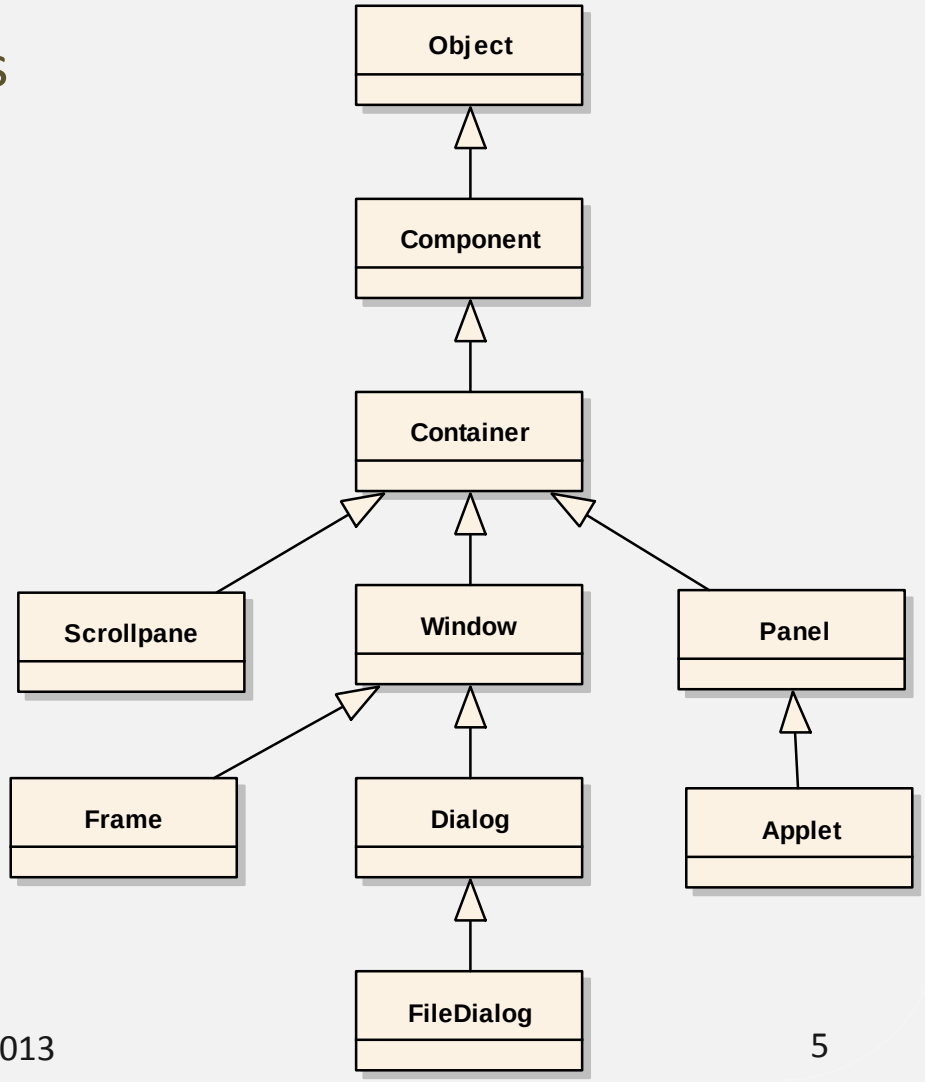
Héritage en analyse

- Cette relation est aussi utilisable en analyse… avec précaution
  - Risque de raisonner par nature…
    - Définir les choses par ce qu'elles sont
    - Par exemple : un chien *est* un animal
    - Ne correspond pas à l'approche « objet »
    - Mêmes problèmes que pour la modélisation d'une voiture
    - Aboutit souvent à des problèmes d'héritage multiple
  - Le raisonnement « objet » est comportemental
    - L'héritage « objet » signifie « se comporte comme… » dans le contexte du système modélisé

Exemple d'arbre d'héritage défini dans Java

- Toute classe Java est une sous classe de la classe Object

- Par exemple Window est une
  - Sous classe de Container
  - Super classe de Frame
  - Super classe de Dialog



Visibilité des opérations, des attributs et des associations

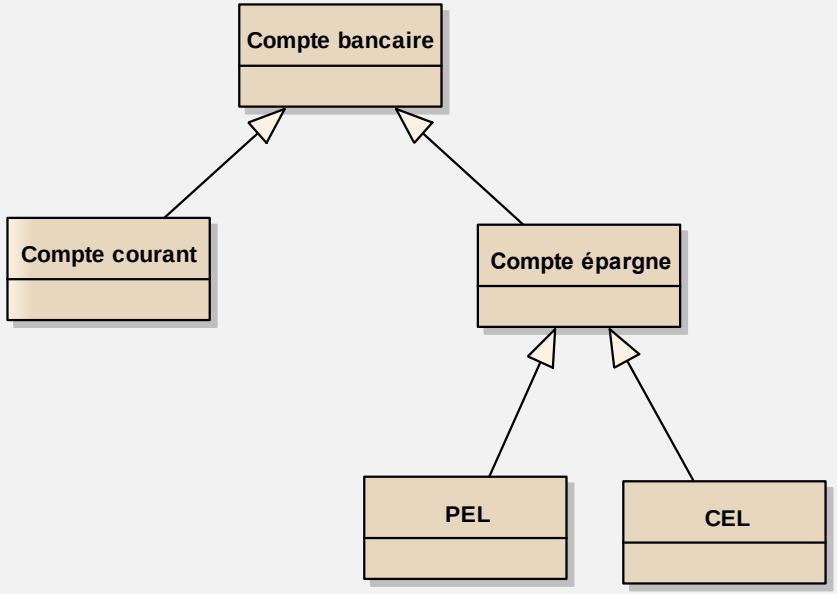
- Les éléments non encapsulées ont une visibilité publique (+)
- Les éléments encapsulées au niveau d'un *package* ont une visibilité *package (~)*
- Les éléments encapsulés au niveau d'un arbre d'héritage ont le plus souvent une visibilité protégée (#) afin d'être visibles par les sous-classes
- Les éléments encapsulés au niveau d'une classe ont une visibilité privée (-)
  - Toujours préférable pour les attributs et les associations (il vaut mieux passer par les getteurs et les setteurs)
  - Possible pour les opérations mais limite l'intérêt de l'héritage
- NB : différence entre UML et Java
  - En Java, la visibilité protégée inclut la visibilité *package*
  - En UML, la visibilité protégée n'inclut pas la visibilité *package*

Utilisations de l'héritage

- Généralisation
  - Factoriser les opérations, les attributs et les relations communs à plusieurs classes au sein d'une super classe
- Spécialisation
  - Réutiliser des opérations existantes en les redéfinissant pour un contexte particulier
  - Réutiliser des attributs et des relations définis dans la super classe (via les getteurs et les setteurs)
- Objectif
  - Réutiliser des portions d'implémentation sans copier-coller (et ainsi éviter les maintenances multiples)

Exemple d'arbre d'héritage dans le domaine bancaire (en analyse)

- Attention au raisonnement par nature lors de l'analyse (cf. TD modélisation d'une voiture)

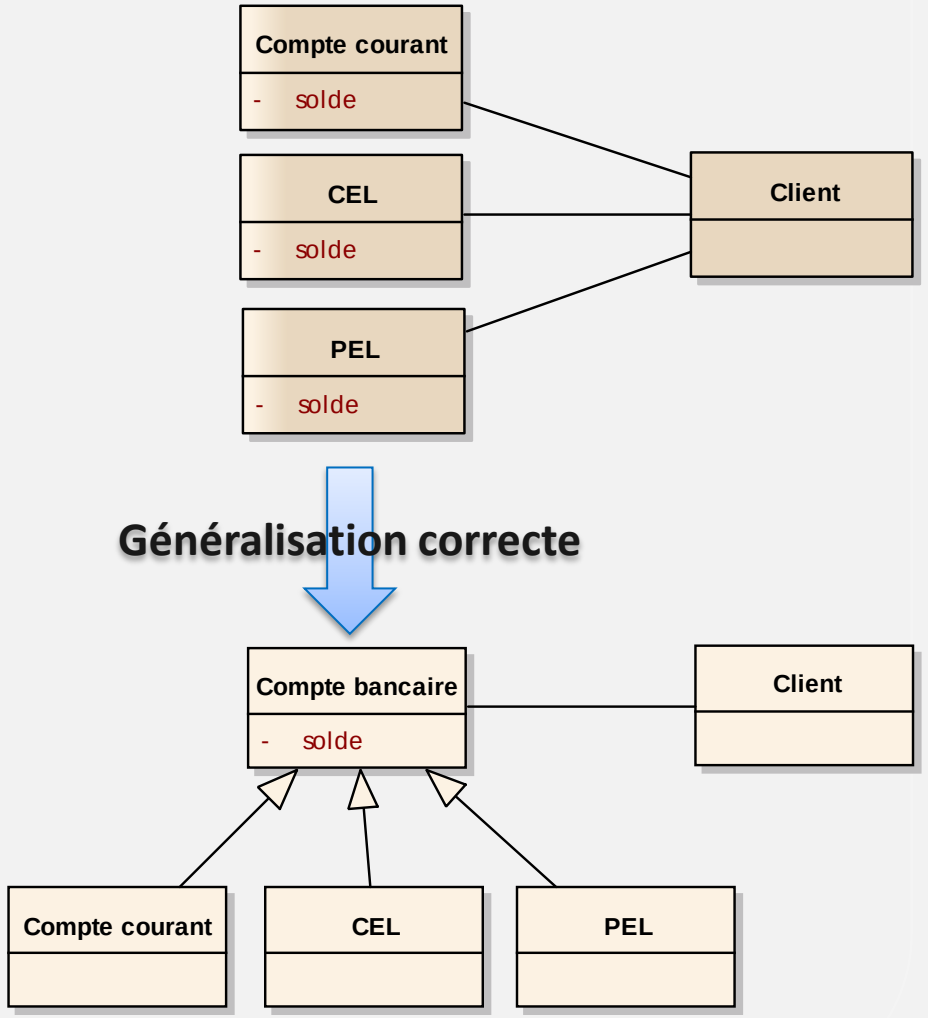


- A ce stade, rien ne garantit que cette structure permettra d'effectuer une factorisation correcte lors de l'implémentation

Un premier niveau de justification
Exemple de généralisation

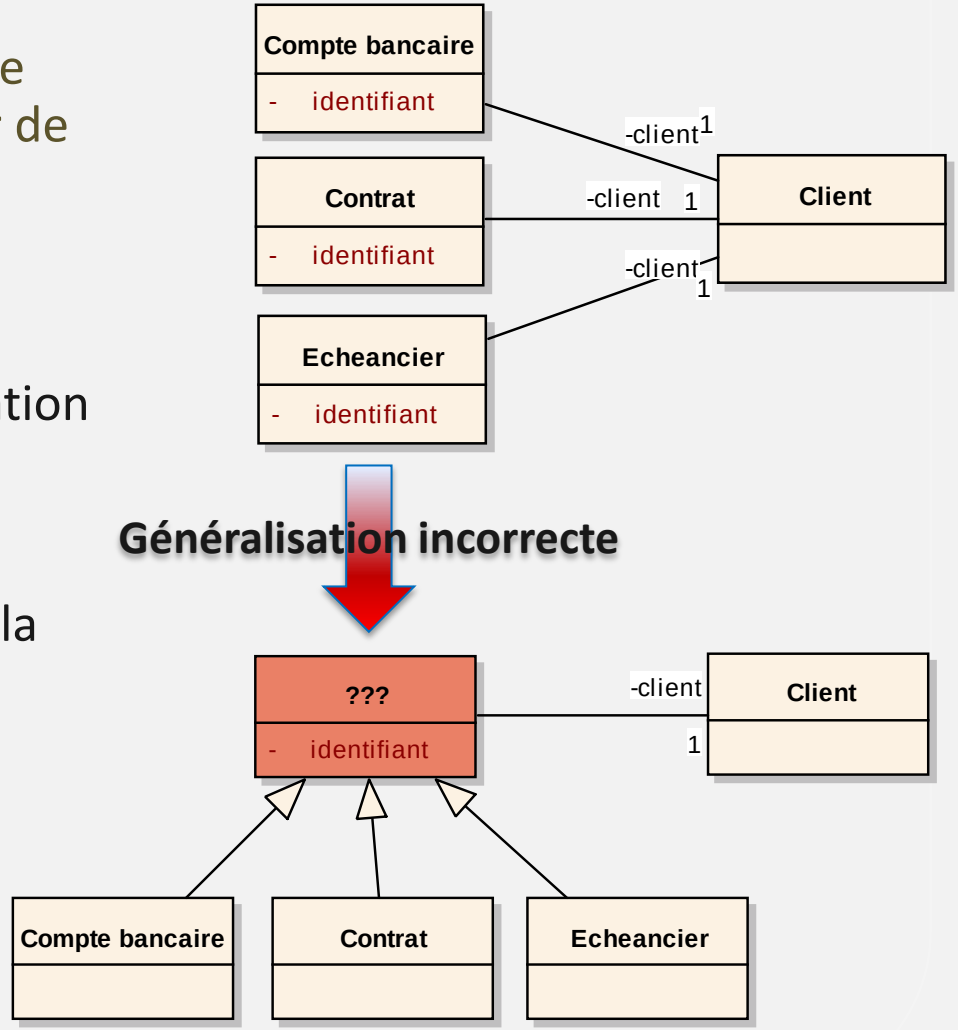
- Sans héritage : l'attribut solde et l'association avec le client sont dupliqués dans plusieurs classes

- La généralisation permet de factoriser dans une classe unique l'attribut solde et l'association avec le client
  - Le solde est défini pour un compte bancaire, un compte courant, un CEL et un PEL
  - L'association avec le client est défini pour ces mêmes éléments



Contre exemple de généralisation

- Factoriser des éléments sur une base purement syntaxique est une erreur de conception
  - Nom des opérations
  - Noms des attributs
  - Noms des rôles pour une association
- Exemple ci-contre
  - 3 classes avec le même lien vers la même classe
  - La généralisation n'a aucun sens
- Que manque t-il ?



Deuxième niveau de justification (plus important que le premier niveau)

- La définition de la responsabilité de chaque sous classe
  - Responsabilité similaire à la super classe
    - Réutilisation de la majorité des opérations
  - Responsabilité spécialisée par rapport à la super classe
    - Redéfinition possible des opérations (cf. page suivante)

- Le respect du comportement attendu dans les diagrammes d'interaction quelle que soit la sous classe

- Le respect du principe d'encapsulation
  - On ne doit pas connaitre l'arbre d'héritage pour pouvoir utiliser les objets issus des classes de cet arbre

Redéfinition / surcharge des opérations

- Si l'opération héritée convient à la sous classe
  - Inutile de redéfinir l'opération héritée
  - L'opération sera exécutée de la même manière que sur la super classe si tous les éléments utilisés par cette opération sont visibles par la sous classe

```
class CompteBancaire … {
 getSolde(){…
  return solde;
  }
```

```
class CompteEpargne … {
 …
```

Redéfinition / surcharge des opérations

- Si l'opération héritée convient partiellement à la sous classe
  - Redéfinition ou surcharge de l'opération héritée
  - En général, une redéfinition commence par appeler l'opération héritée pour garantir l'évolution du code (le mot clé « super » en Java permet d'accéder à la super classe)

```
class CompteBancaire … {
 calculerInterets(){…
  return 0;
  }
```

```
class CompteEpargne … {
 calculerInterets(){…
  return super.calculerInterets()+solde*0.75/100;
  }
```

Redéfinition / surcharge des opérations

- Si l'opération héritée ne convient pas à la sous classe
  - Redéfinition ou surcharge de l'opération héritée sans appeler l'opération héritée
  - (Revoir l'arbre d'héritage si ce cas se répète trop souvent)

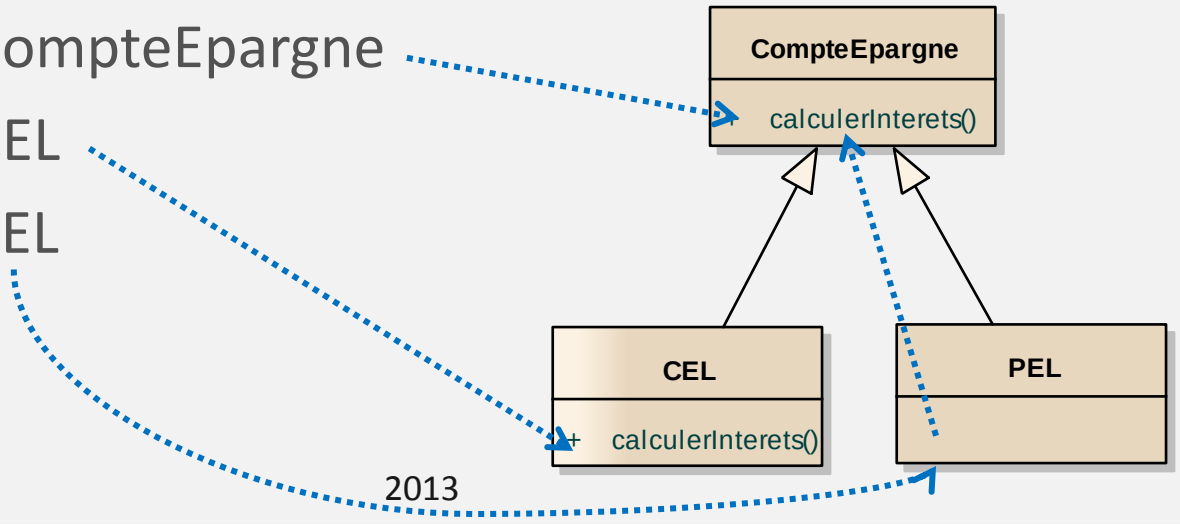
```
class CompteBancaire … {
 calculerInterets(){…
  return 0;
  }
```

```
class CompteEpargne … {
 calculerInterets(){…
  return solde*0.75/100;
  }
```

Recherche d'une opération dans l'arbre d'héritage

- Recherche de l'opération correspondant au message reçu
  - Dans la classe de l'objet receveur (comme d'habitude)
  - Si l'opération n'est pas définie dans cette classe, recherche dans la super classe (jusqu'à la classe racine, Object en Java)

- Exemple : réception d'un message calculerInterets par…
  - Un objet :CompteEpargne
  - Un objet :CEL
  - Un objet :PEL



Héritage et constructeur

- Bonne pratique
  - Le constructeur d'une sous-classe commence par appeler le constructeur de la super classe pour initialiser les attributs hérités
  - Si vous ne le faites pas, Java appelle implicitement un constructeur sans argument

Un peu de théorie…
Postulat de Barbara Liskov

- Coté implémentation
  - Une sous-classe doit être conçue de sorte que ses instances puissent se substituer à des instances de la classe de base partout où cette classe de base est utilisée
- Coté utilisation
  - Ceux qui utilisent des objets d'une classe doivent pouvoir utiliser des objets d'une sous-classe sans même le savoir
- De mon point de vue
  - C'est l'application du principe d'encapsulation à un arbre d'héritage

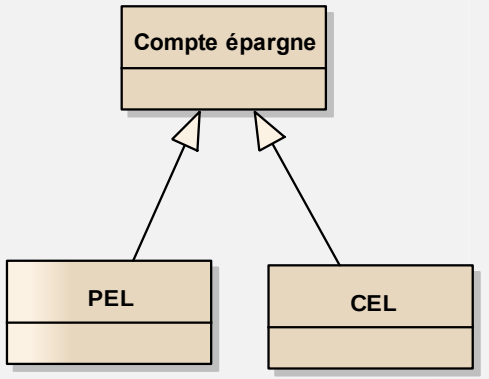
En pratique…

- Commencer sans utiliser l'héritage (analyse, conception, code)

- Détecter les héritages à créer à l'aide d'un truc « infaillible »…
  - Si vous identifiez plusieurs fois les mêmes cascades de tests conditionnels
    - Dans des règles de gestion en analyse,
    - Dans les définitions des opérations en conception,
    - Ou lors du codage…
  - Vous pouvez créer un arbre d'héritage pour encapsuler ces cascades conditionnelles
  - Le mécanisme qui permet cela s'appelle le polymorphisme

En pratique : un truc « infaillible »
Une illustration

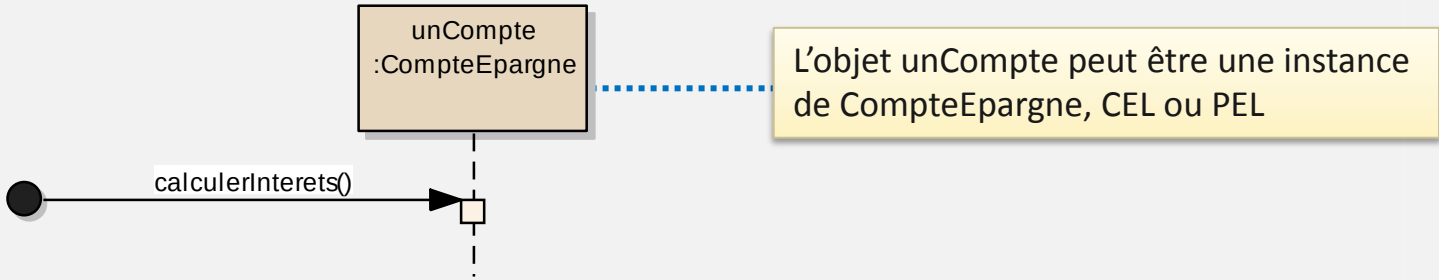
- Exemple 1 : calcul des intérêts sur les comptes bancaires
  - Si le compte est un PEL alors intérêts = 2.5% …
  - Sinon si le compte est un CEL alors intérêts = 0.75% …
  - Sinon intérêts = 0
- Exemple 2 : calcul du coût d'un prêt utilisant un compte épargne
  - Si le compte est un PEL alors …
  - Sinon si le compte est un CEL alors …
  - Sinon …
- Ces deux exemples justifient l'arbre d'héritage ci-contre



Le polymorphisme permet d'encapsuler les cascades conditionnelles

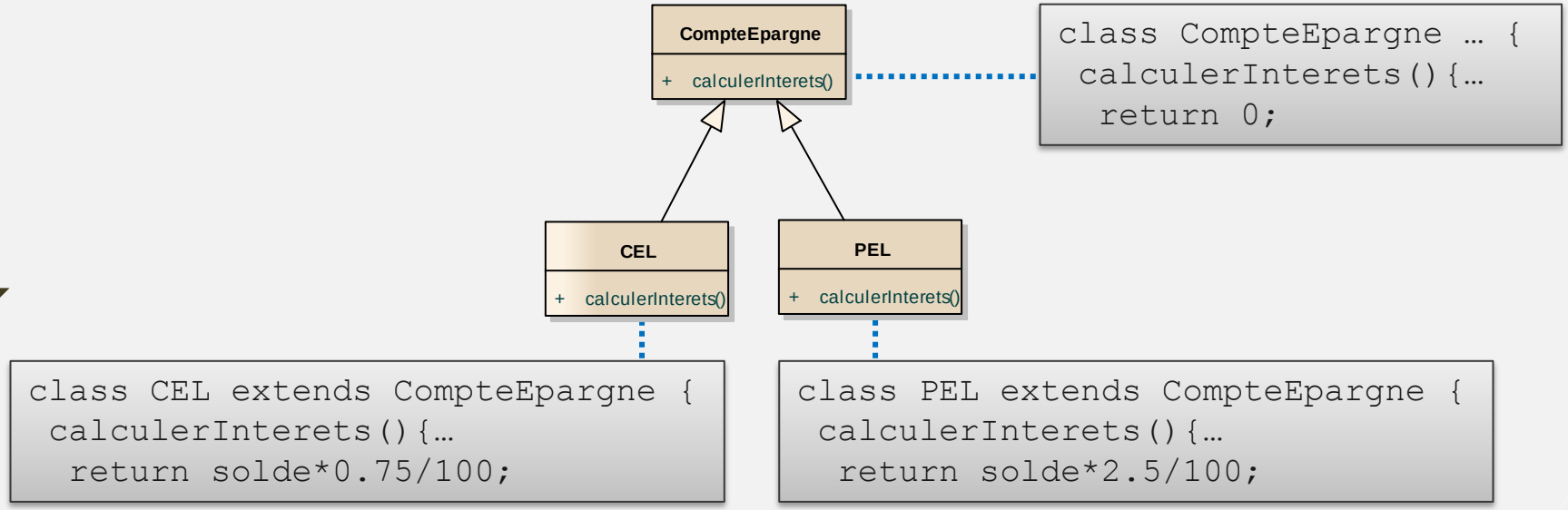
aspects dynamiques

- Faculté d'objets de classes différentes à répondre à un même message



aspects statiques

- Le résultat de l'opération dépend de l'objet receveur du message



Impact du polymorphisme sur la maintenance du code

- Sans polymorphisme
  = cascades conditionnelles

```
class Exemple{
 … { …
  if (unCompte instanceof…) {
    …}
  elseif (unCompte instanceof…) {
    …}
  elseif (unCompte instanceof…) {
    …}
  elseif (unCompte instanceof…) {
    …}
  …
```

- Avec polymorphisme
  = un arbre d'héritage

```
class Exemple {
 … { …
  unCompte.calculerinterets();
…
```

```
class CompteEpargne … {
 calculerInterets(){…
  return 0;
```

```
class CEL extends CompteEpargne {
 calculerInterets(){…
  return solde*0.75/100;
```

```
class PEL extends CompteEpargne {
 calculerInterets(){…
  return solde*2.5/100;
```

Impact du polymorphisme
Exemple : ajout d'une classe LivretA

- Sans polymorphisme
  = impacts multiples…

```
class Exemple{
 … { …
  if (unCompte instanceof…) {
    …}
  elseif (unCompte instanceof…) {
    …}
  elseif (unCompte instanceof…) {
    …}
  elseif (unCompte instanceof…) {
    …}
  elseif (unCompte instanceof…) {
    …}
  …
```

… sur chaque cascade conditionnelle

- Avec polymorphisme
  = pas d'impact sur l'existant

```
class Exemple {
 … { …
  unCompte.calculerinterets();
…
```

```
class CompteEpargne … {
 calculerInterets(){…
  return 0;
```

```
class CEL extends CompteEpargne {
 calculerInterets(){…
  return solde*0.75/100;
```

```
class PEL extends CompteEpargne {
 calculerInterets(){…
  return solde*2.5/100;
```

```
class LivretA extends CompteEpargne {
 calculerInterets(){…
  return solde*1.25/100;
```

Synthèse sur l'héritage

- Outil très puissant évitant de nombreux copier/coller de code
  - Eviter de définir des arbres d'héritage trop tôt
  - Détecter les cascades conditionnelles répétées

- Ne pas confondre interface et héritage
  - Interface = encapsulation = adhérence faible entre client et implémentation
    - Cycles d'évolution indépendants
  - Relation d'héritage = relation d'implémentation
    = adhérence forte entre classes de l'arbre d'héritage
    - Cycles d'évolution liés

L'héritage ne se décrète pas, il se construit progressivement

- L'arbre d'héritage se construit au fur et à mesure (de l'analyse,) de la conception et du développement
  - Pour chaque nouvel attribut, association ou opération, il faut le/la positionner au bon endroit dans l'arbre d'héritage
  - Il ne faut pas hésiter à modifier l'arbre si nécessaire

- Impacts sur la maintenance
  - Une modification sur une classe aura un impact sur toutes les sous classes
    - Pas besoin de copier/coller les modifications
    - Mais il faut mesurer les impacts sur les sous classes
  - Une modification sur une classe n'aura pas d'impact sur les super classes

Classe abstraite

- Classe ne pouvant pas créer d'objet
- Classe utilisée pour factoriser, via un héritage, les opérations, les attributs et les relations communs à plusieurs classes
- Peut contenir en plus des opérations abstraites (= seulement la signature comme les interfaces)
  - Les sous classes doivent redéfinir ces opérations
- NB : une classe abstraite est inutile hors héritage