Conception objet

Principes, pratiques et mise en œuvre

DUT Informatique
Semestre 3

Mourad Ouziri
mourad.ouziri@parisdescartes.fr

Université Paris Descartes

Plan du cours

- Objectif : maintenabilité des applications à objets
- Plan
  - Principes de conception objet et refactoring de code
  - Etude de quelques bonnes pratiques de conception « Design pattern »
- Intervenants :

  M. Ouziri, Mikal Ziane
- Evaluation :

  1 DST (coeff 3) et 1 TP noté (coeff 1)

Etude de quelques principes de conception objet

Références bibliogtaphiques

- *Software Architecture in Practice (2nd edition) – Bass, Clements, Kazman, Addison-Wesley 2003*
- *Agile Software Development: Principles, Patterns, and Practices – R. C. Martin, Prentice Hall 2003*

Motivation

- Objectif : développer des applications de qualité
  - Plusieurs critères de qualité (ISO 9126, FURPS, etc.)
- Maintenabilité, un critère de qualité important !
- Quelques définitions
  - Les changements qui doivent être apportés à un logiciel après sa livraison à l'utilisateur *(Martin et McClure 1983)*
  - La totalité des activités qui sont requises afin de procurer un support, au meilleur coût possible, d'un logiciel. Certaines activités débutent avant la livraison du logiciel, donc pendant sa conception initiale (*SWEBO 2005*)

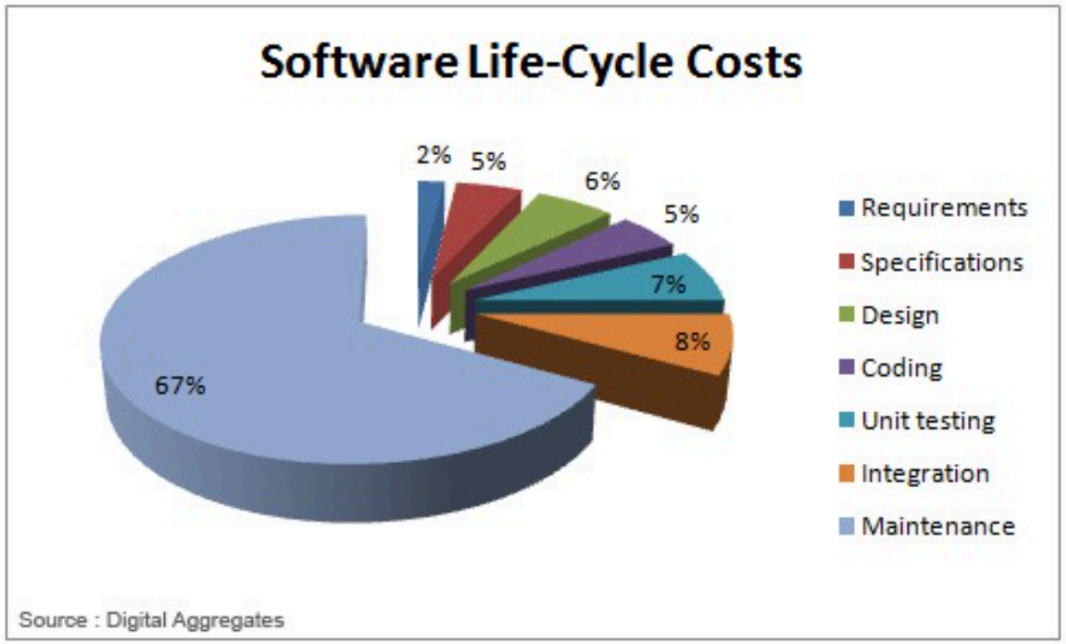
Motivation

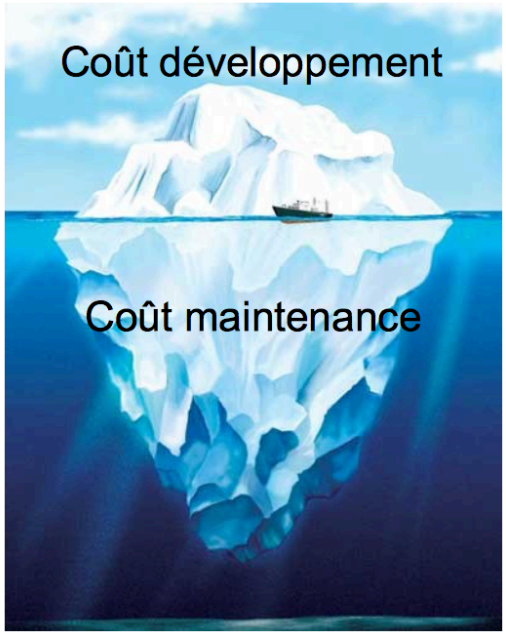
☞ Maintenabilité, un facteur de qualité important !

– Evolution constante des attentes des utilisateurs et des technologies

– Coût moyen des applications très élevé

– Amortissement du coût de développement

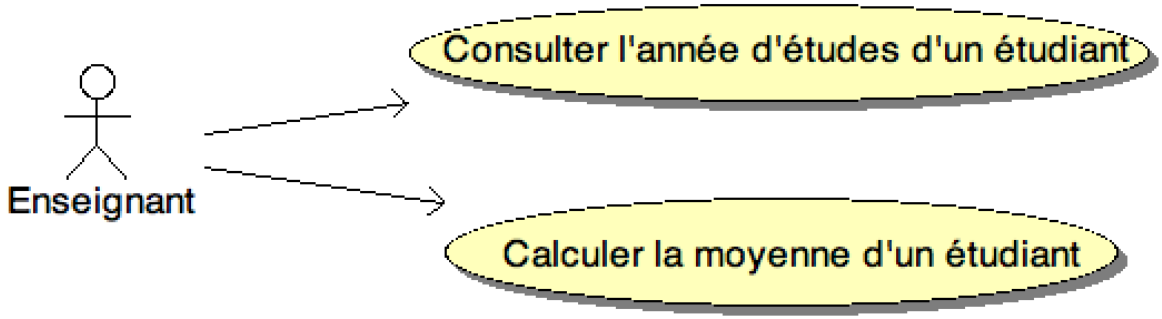


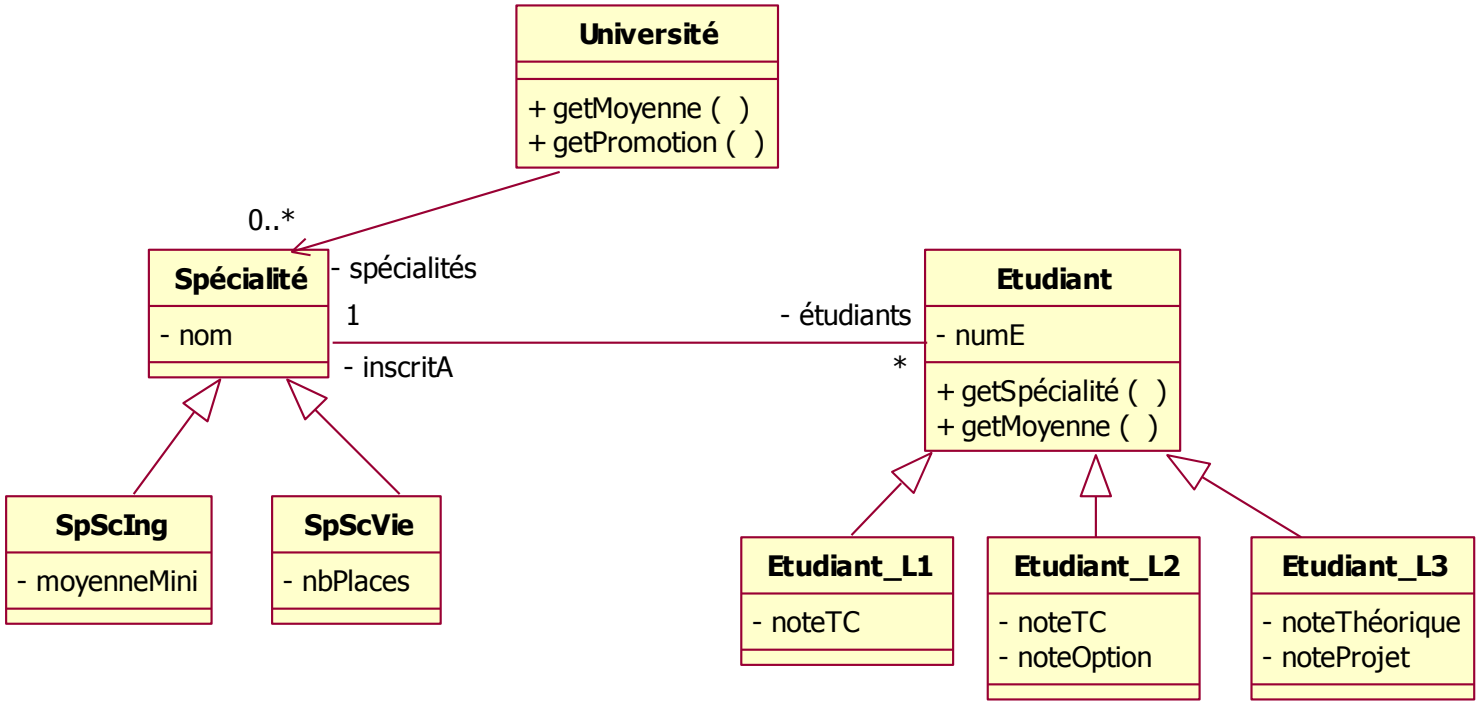


☞ Objectif : réduire les coûts des maintenances

Analyse de maintenabilité

☞ Soit l'application suivante





Analyse de maintenabilité

Redondance de code

☞ Application « Université » : consulter l'année d'études d'un étudiant

```
public class Université1 {
    public String getAnneeEtudes (Etudiant  etud) {
      if (etud.getClass().getName() == "Etudiant_L1")  return "Licence 1ère année" ;
      if (etud.getClass().getName() == "Etudiant_L2")  return "Licence 2ème année";
      if (etud.getClass().getName() == "Etudiant_L3")  return "Licence 3ème année";
    }
    public String getAnneeEtudes (Long  numE) {
       for (Spécialité sp : spécialités) {
          List<Etudiant>  etuds = sp.getEtudiants ();
          for (Etudiant etud : etuds) {
                  if (numE == etu.getNuméro ())
                      if (etud.getClass().getName() == "Etudiant_L1")  return "Licence 1ère année" ;
                      if (etud.getClass().getName() == "Etudiant_L2")  return "Licence 2ème année";
                      if (etud.getClass().getName() == "Etudiant_L3")  return "Licence 3ème année";
          }
      }}}
```

Analyse de maintenabilité

Responsabilité de classes

☞ Application « Université » : calculer la moyenne d'un étudiant (1)

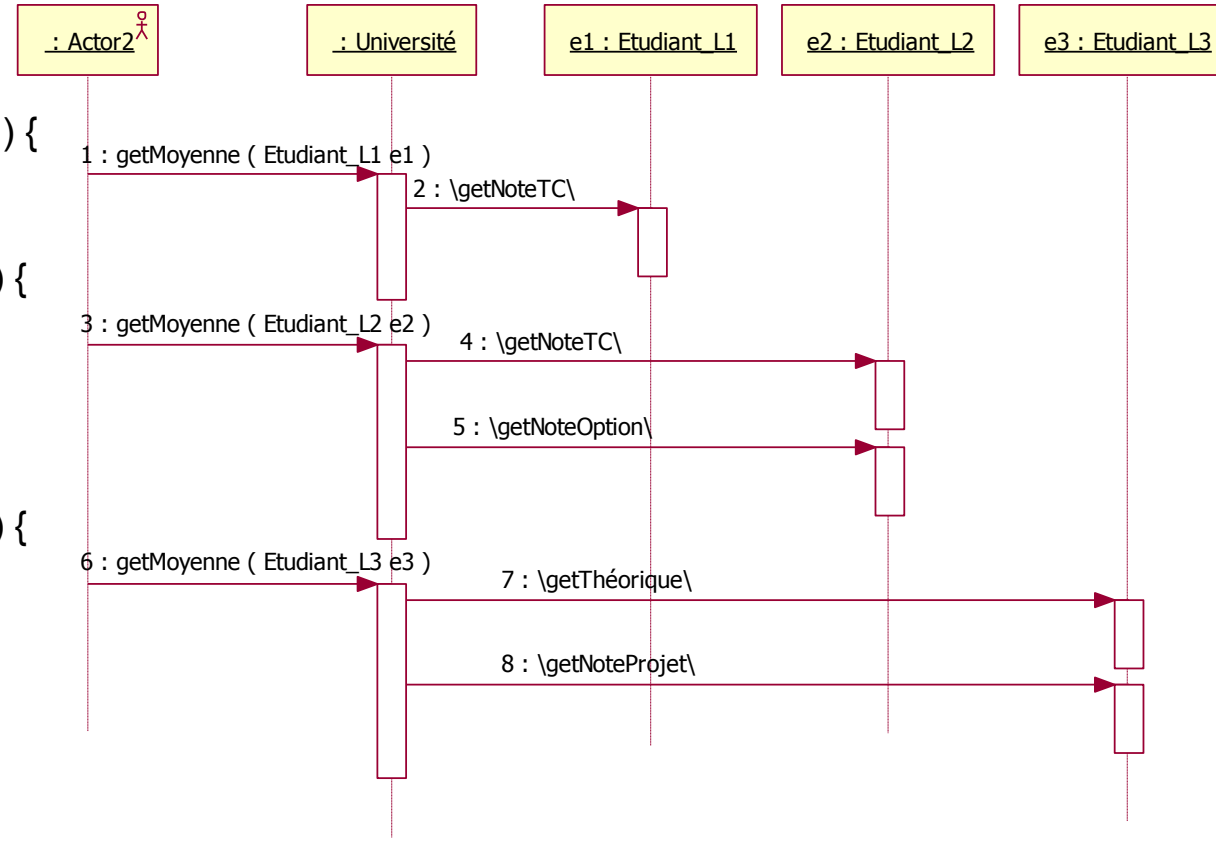
```
public class Université2 {
    private List<Etudiant> etuds ;
    public double getMoyenne (int numeEtud) {
      Etudiant etud = this.rechecherEtudiant (numEtud);
      if (etud.getClass().getName().equals( "Etudiant_L1")) {
            Etudiant_L1 el1 = (Etudiant_L1) etud;
            return el1.getNoteTC(); ;
      }
      else if (etud.getClass().getName().equals("Etudiant_L2")) {
           Etudiant_L2 el2 = (Etudiant_L2) etud;
           return (el2.getNoteTC() + el2.getNoteOption()) / 2;
      }
      …
}}
```

Analyse de maintenabilité

Responsabilité de classes

☞ Application « Université » : calculer la moyenne d'un étudiant (2)

```
public class Université2 {
   float getMoyenne (Etudiant_L1 etud) {
     return etud.getNoteTC();
   }
   float getMoyenne (Etudiant_L2 etud) {
     return (etud.getNoteTC() +
             +etud.getNoteOption())
              /2;
   }
   float getMoyenne (Etudiant_L3 etud) {
     return (etud.getNoteThéorique()
             +etud.getNoteProjet())
             / 2;
   }
}
```



Analyse de maintenabilité

Evolution

☞ Evolutions

(1) Gérer de nouvelles promotions (*Master 1ère* et *2ème année*)

(2) La note de tronc commun en *L2* se décline désormais en une note des matières de la spécialité et une note des matières générales

(3) Un stage obligatoire doit être suivi par les L3

☞ Coûts de ces évolutions

(1) Coder les nouvelles classes *Etudiant_M1* et *Stage* (OK, coûts directs)

Modifier les classes *Université* (deux méthodes de la classe *Université*) ! (KO, coûts indirects, multiplication de l'effort !)

(2)(3) Ajout d'attributs pour les nouvelles notes dans les classes respectives *Etudiant_L2* et *Etudiant_L3* (OK, coûts directs)

Modifier la classe *Université* (KO, coûts indirects !)

Analyse de maintenabilité

Evolution

☞ Evolution (1) : écrire **les classes *Master*** et **modifier (à 2 endroits) la classe**

Université

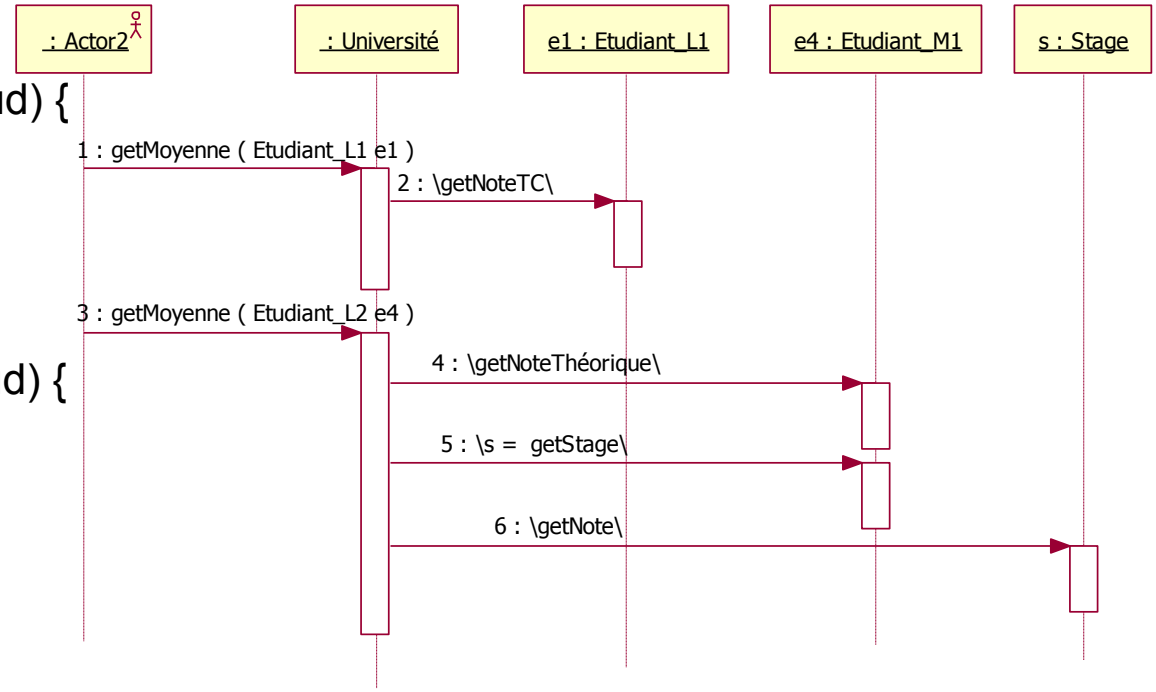
```
class Université {
    String getPromotion (Etudiant  etu) {
      if (etud.getClass() == "Etudiant_L1")  return "Licence 1ère année" ;
      …
      if (etud.getClass() == "Etudiant_M1")  return "Master 1ère année" ;          // Evolution !!!
    }
    String getPromotion (Long  numE) {
       for (Spécialité sp : spécialités) {
          List<Etudiant> etuds = sp.getEtudiants ();
          for (Etudiant etu : etuds) {
                    if (numE == etu.getNuméro ())
                       if (etud.getClass() == "Etudiant_L1")  return "Licence 1ère année" ;
                       …
                       if (etud.getClass() == "Etudiant_M1")  return "Master 1ère année" ;// Evolution !!!
          }
      }}}
```

Analyse de maintenabilité

Evolution

☞ Evolution (1) : écrire **les classes *Master*** et **modifier la classe Université**

```
class Université {
   float getMoyenne (Etudiant_L1 etud) {
     return (etud.getNoteSpec()+
              etud.getNoteGle())/2
   }
…
   float getMoyenne (Etudiant_M1 etud) {
     Stage st = etud.getStage();
     return (etud.getNoteThéorique()
            +st.getNote())
            / 2;
   }
}
```



☞ Evolution : et si la classe *Stage* évoluait ?!

- Elle devient : *StageEntreprise (intitulé, noteEntrep)* et *Soutenance (obsJury, noteSout)*
- Quels impacts sur la classe *Université* ?

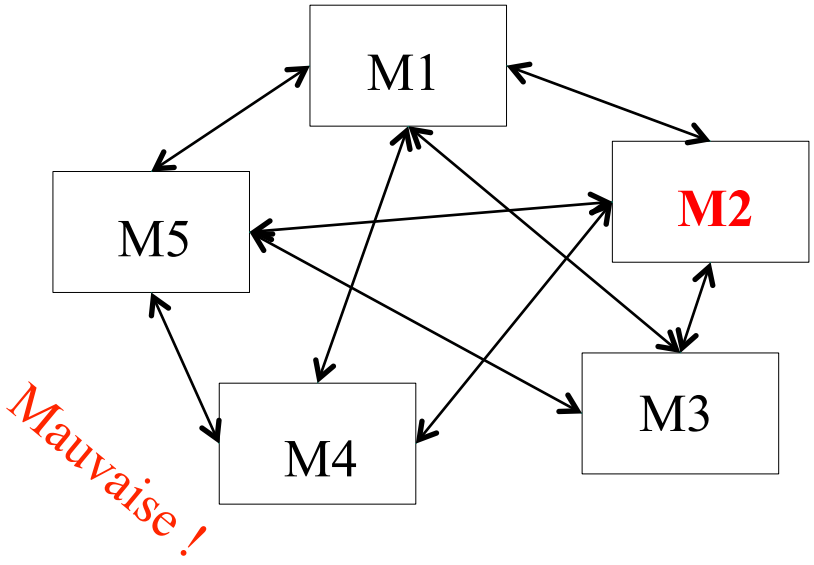
Architecture des applications

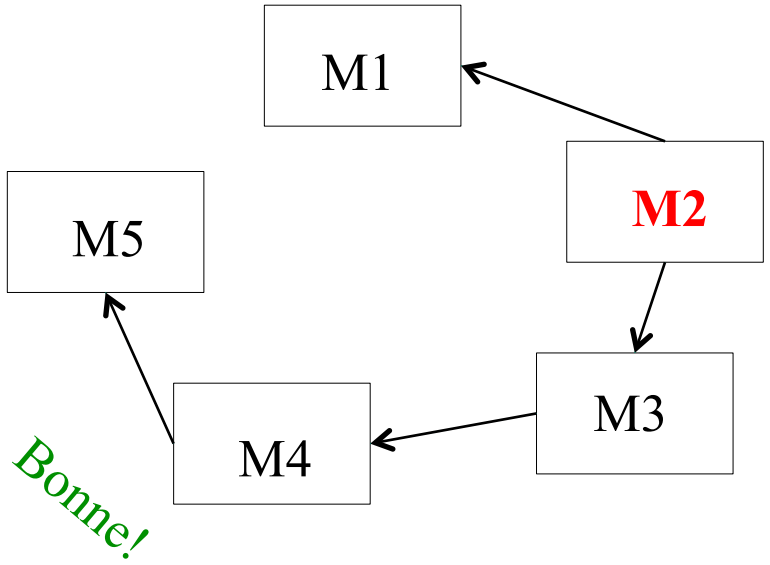
- Définition
  - Architecture = Structure de l'application
- Eléments d'architecture
  - Composants : modules (classes, packages, modules fonctionnels) constituant l'application
  - Rôles des composants
  - Communication inter-composants
- Objectif
  - Concevoir des applications robustes vis-à-vis des évolutions futures
  - Concevoir des applications maintenables à moindre coût

Architecture des applications

☞ Schéma d'architecture

Légende :
M1 → M2 : M1 fait appel à M2





☞ Problèmes de maintenabilité

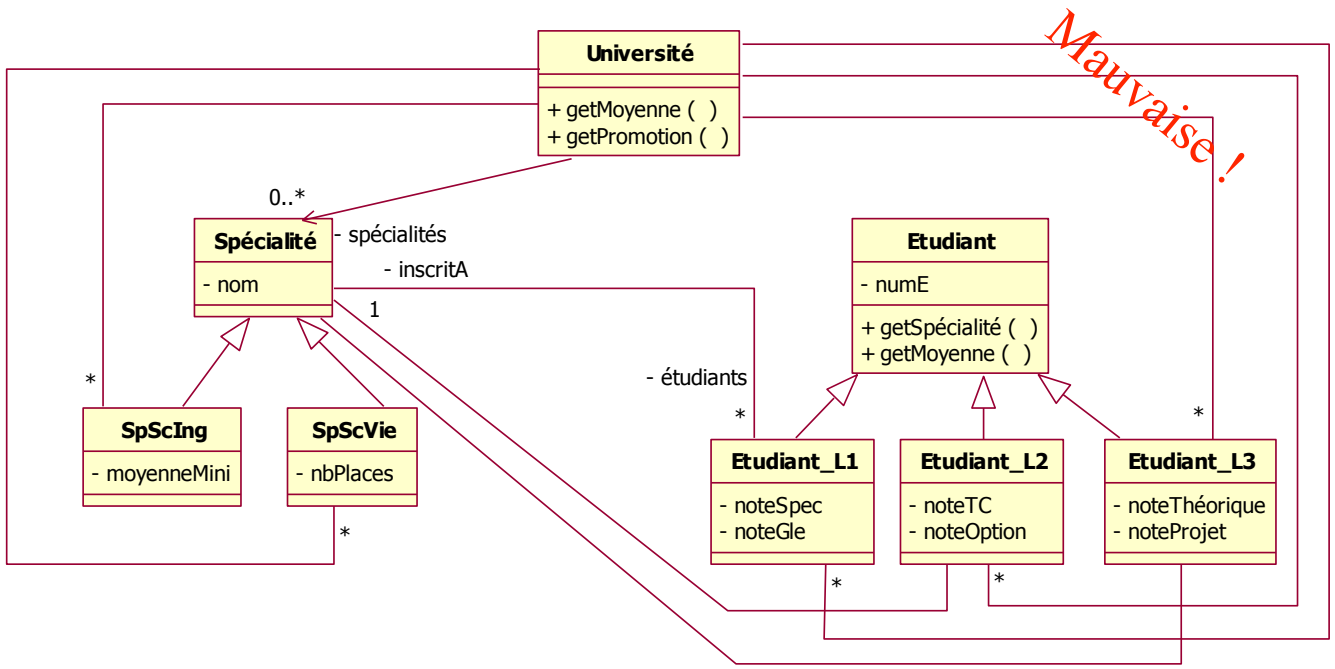
– Propagation (impact) du changement (des structures de données par exemple)

– Coûts = Coûts liés à M1 + Coûts des modules impactés par propagation (M2, M3, M4, M5)

Architecture des applications

☞ Architecture des applications à objets

Dépendances structurelles : relations structurelles entre classes



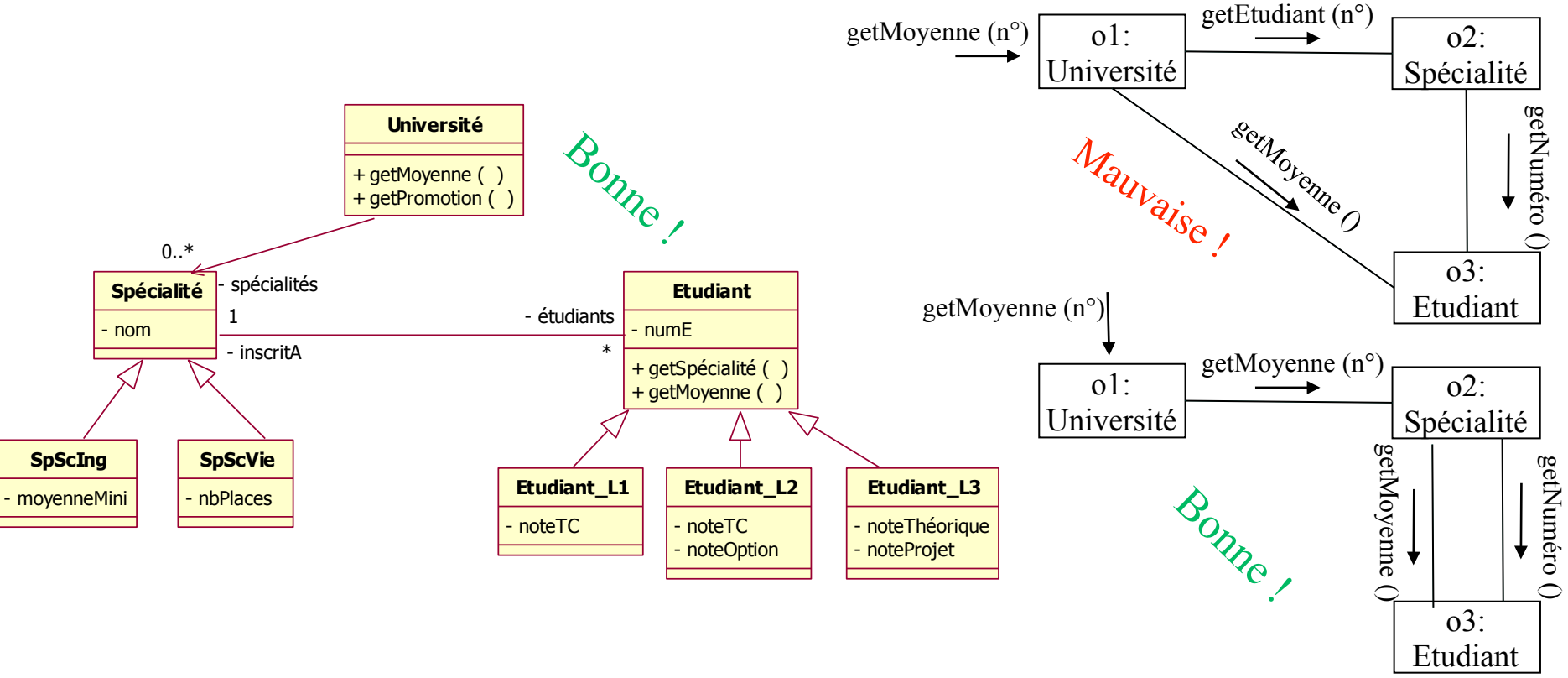

```
class Université {
    List<Etudiant_L1>  etudsL1;
    List<Etudiant_L2>  etudsL2;
    List<Etudiant_L3>  etudsL3;
    …
}

class Spécialité {
    List<Etudiant_L1>  etudsL1;
    List<Etudiant_L2>  etudsL2;
    List<Etudiant_L3>  etudsL3;
…
}
```

☞ La maintenabilité dépend des dépendances structurelles entre classes

Architecture des applications

- Architecture des applications à objets

  Dépendances dynamiques : appels de méthodes inter-objets

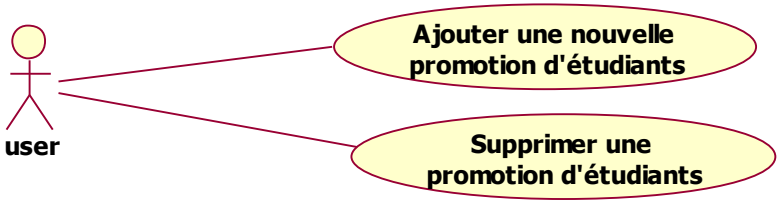


- La maintenabilité dépend des dépendances comportementales entre classes

Architecture de classes

☞ Exercice : évolution de l'application « Université »

L'utilisateur souhaite ajouter et supprimer des promotions d'étudiants pendant l'exécution



☞ A faire !

Intégrer cette nouvelle demande au diagramme de classes

Recoder le méthode *getPromotion (Etudiant e)* de la classe *Université*

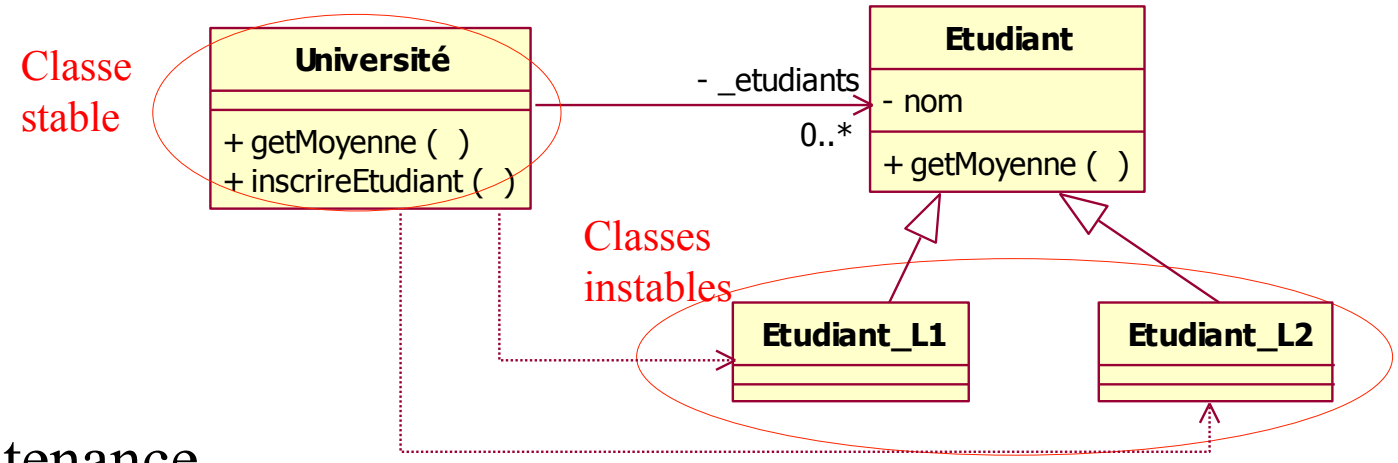
☞ Conclusion : quel est l'impact de cette évolution ?

(1) Nombre de classes ajoutées (OK)

(2) Nombre de classes modifiées (KO!)

Facteurs de maintenabilité

- Classes (package, modules) instables ou évolutifs :
  - Classes pouvant être modifiées lors des opérations de maintenance
- Dépendance amplifiant l'effort de maintenance :
  - Dépendance de classes supposées stables vers des classes supposées instables



- Coûts d'une maintenance
  - Coûts directs (liés à la maintenance) et coûts indirects (effet de bord, induit par propagation via les dépendances)
  - Le travail de maintenance vise à minimiser les coûts indirects !

Résumé de ce qui amplifie la maintenance

- Dépendance de classes qui risquent d'évoluer (dites « instables »)
  - Dépendance d'implémentations « instables »
  - Remède : ne pas faire dépendre de classes instables

- Attribution des rôles
  - Attribution non judicieuse des rôles aux classes
  - Remède : limiter les traitements des méthodes aux seules données/attributs de la classe (respect des rôles)
- Redondance de code
  - La redondance de code multiplie le travail de maintenance
  - Remède : factoriser le code redondant dans des super-classes « abstraites »
- Architecture en « spaghetti »
  - Dépendances anarchiques entre classes
  - Remède : structurer en « lasagne » voire linéaire

Architecture de classes
Objectif

- Objectif
  - Minimiser l'effort (coût) nécessaire à la réalisation des maintenances
- Comment ?
  - Limiter la propagation des changements (d'évolutions ou de corrections de bugs)
- Concevoir une bonne architecture de maintenabilité
  - Définir les classes de l'application
  - Définir judicieusement le rôle de chaque classe (attributs et méthodes)
  - **Eliminer toute dépendance vers de classes *instables***
- Classe instable
  - **Classe qui risque de changer lors d'une opération de maintenance**

Règles de conduite

Quelques principes de conception

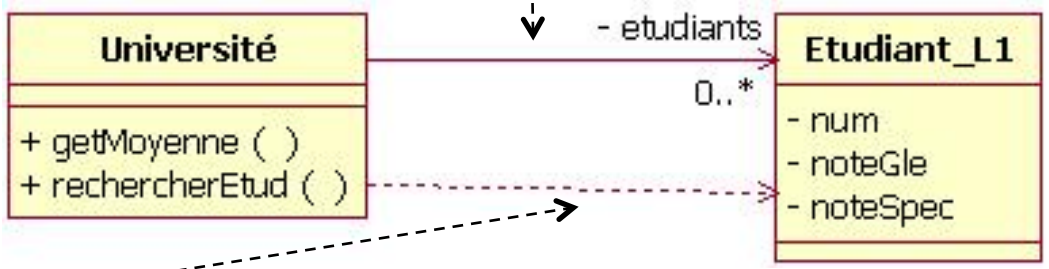
- Considérer la maintenabilité dès la phase de conception initiale
- Inversion de dépendance
  - Faire dépendre d'entités abstraites (supposées stables) plutôt que de classes concrètes (à implémentations souvent instables)
  - Permet d'éliminer les dépendances structurelles amplifiant les maintenances
- Délégation à l'expert d'information
  - Déléguer (transférer) les traitements aux classes responsables
  - Permet d'éliminer les dépendances de comportement nuisibles à la maintenabilité
- Principe de l'Open-Closed (OCP)
  - *Ouvert* aux extensions : intégrer les évolutions par ajout de classes
  - *Fermé* aux modifications : intégrer les évolutions sans modification de classes existantes (conçues, réalisées, testées, validées)
- Responsabilité limitée :
  - Il ne devrait y avoir qu'une raison provoquant le changement dans une classe
- Forte cohésion : une classe doit avoir des responsabilités cohérentes

Dépendances

Structurelles et comportementales

```
class Université {
      List<Etudiant_L1> etudiants;
      float getMoyenne (Etudiant_L1 etud) {
            return (etud.getNoteSpec()+
                        etud.getNoteGle())/2
      }
      String getPromotion (int num) {
          Etudiant etud = recheEtud (num);
            if (etud.getClass() == "Etudiant_L1") …
      }
}
```

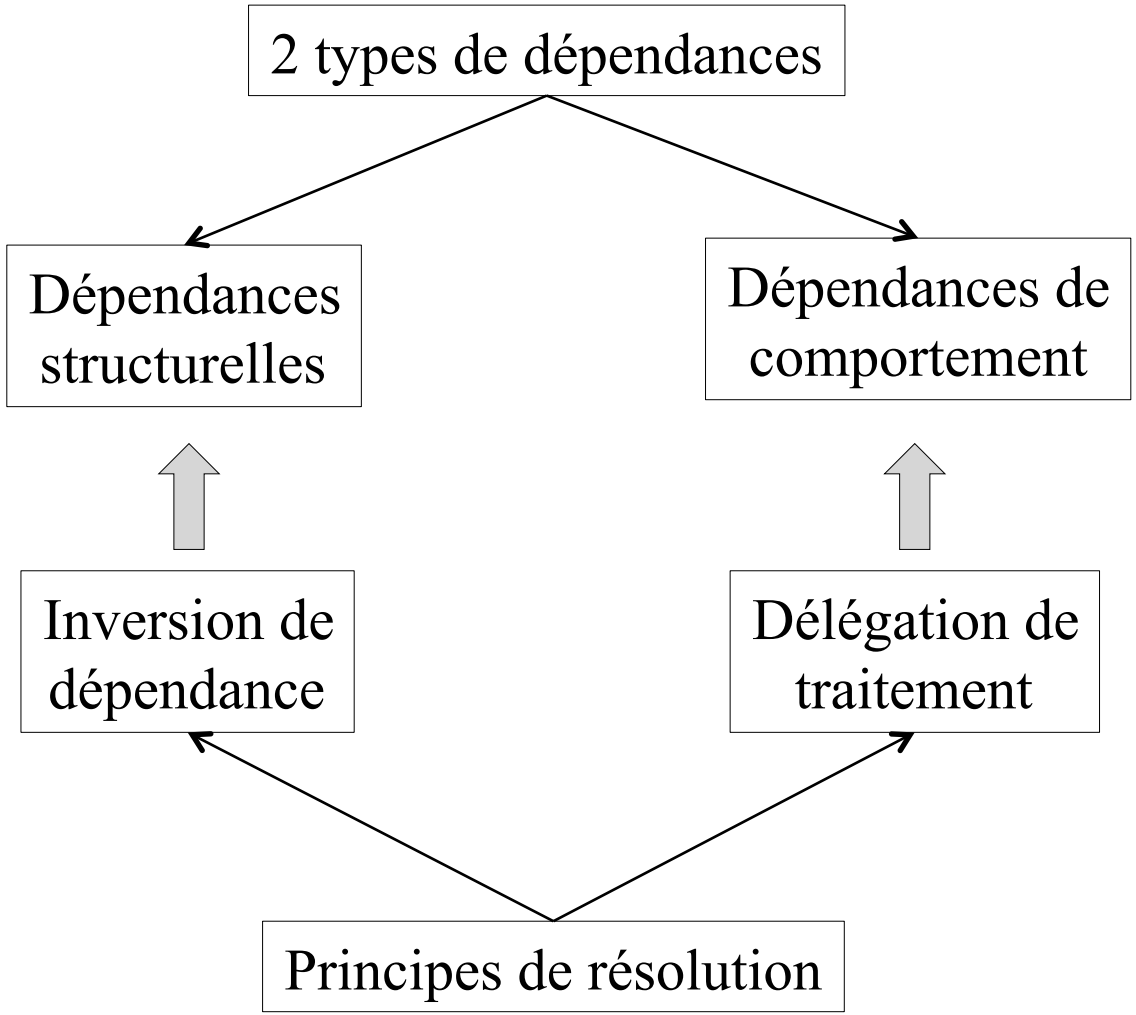
dépendance structurelle



dépendance comportementale

Résolution des dépendances

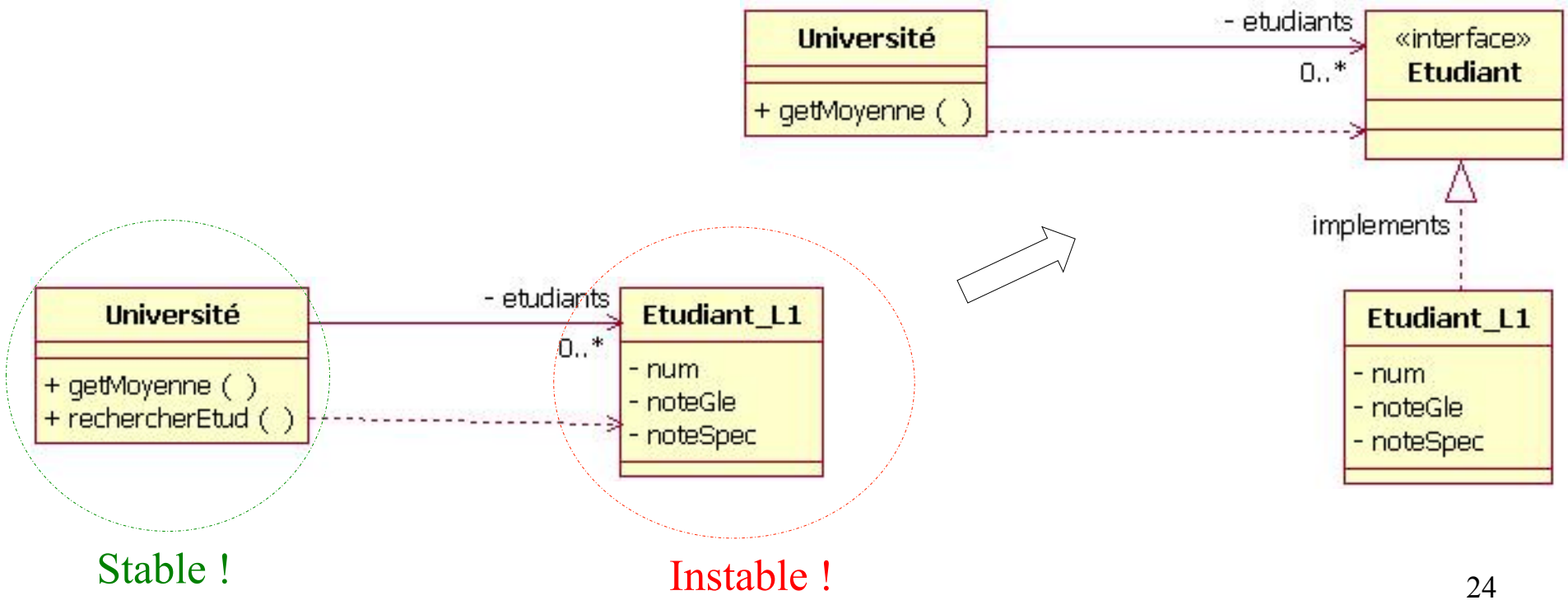
Structurelles et comportementales



Principes de conception
Inversion de dépendance

☞ Inversion de dépendance

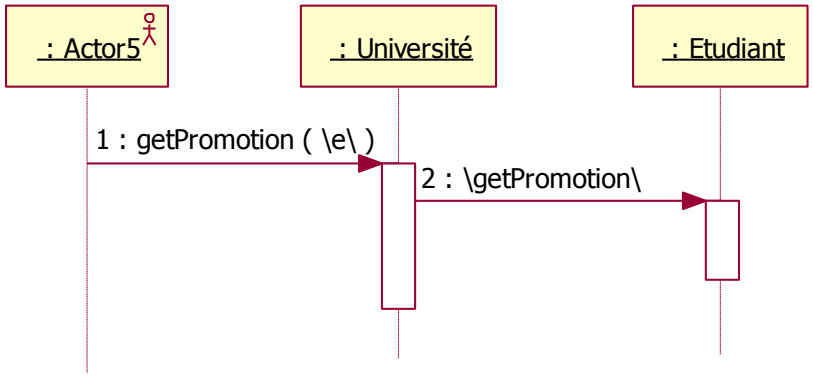
- But : éliminer les dépendances structurelles vers des classes instables
- Mécanisme : faire dépendre d'entités abstraites (interface ou classe abstraite) supposées « stables » créées à partir de classes concrètes jugées « instables »



Principes de conception

☞ Délégation de traitements : respect des rôles !

- But : éliminer les dépendances de comportement
- Mécanisme : respect des rôles en transférant/délégant les traitements aux classes adéquates (expertes d'information)




```
class Université {
      String getPromotion (Etudiant  etud) {
          return etud.getPromotion () ;
      }
}
```

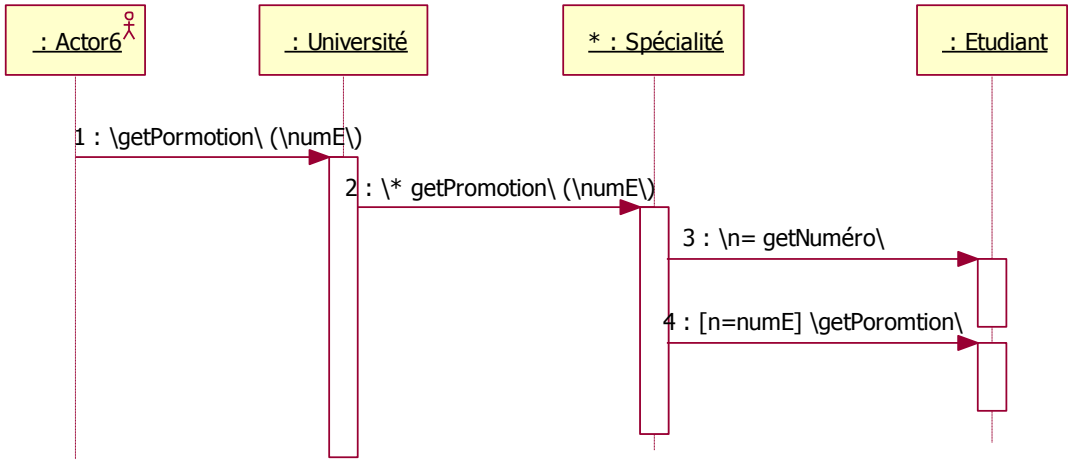
```
class Etudiant_L1 extends Etudiant {
      String getPromotion () {
          return "Licence 1ère année" ;
      }
}
```

```
class Etudiant_L2 extends Etudiant{
      String getPromotion () {
          return "Licence 2ème année" ;
      }
}
```

```
class Etudiant_L3 extends Etudiant{
      String getPromotion () {
          return "Licence 3ème année" ;
      }
}
```

Principes de conception

☞ Application « Université » : **avec délégation…**




```
class Université {
      String getPromotion (Long umE) {
          for (Spécialité sp : spécialité) {
                return sp.getPromotion (numE);
          }
          return null;
}}
```

```
class Spécialité {
  public String getPromotion (Long numE) {
     Etudiant etud = this.rechercheEtud (numE);
     if (etud != null)
         return etud.getPromotion ();
     throw new Exception("etudiant inconnu !");
}}
```

```
class Etudiant_L1 extends Etudiant {
      String getPromotion () {
           return "Licence 1ère année" ;
      }
}
```

Principes de conception

☞ Application « Université » : évolution, ajout de la classe *Etudiant_M1*

– Principe « Open-closed » :

(1) Conception de la nouvelle classe *Etudiant_M1* (OK)

(2) Aucune modification apportée aux classes *Université* et *Spécialité* (++)

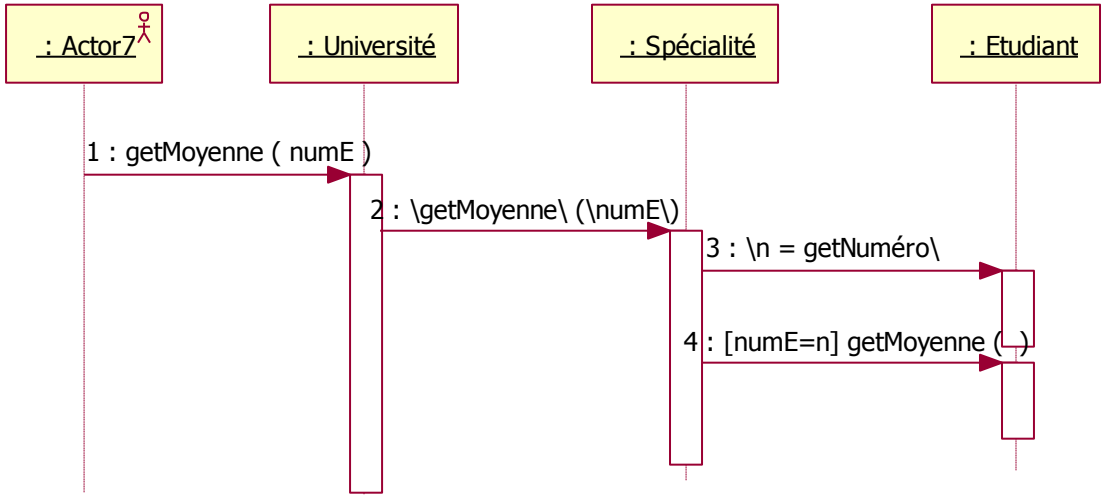
```
class Université {
      String getPromotion (Etudiant  etud) {
          return etud.getPromotion () ;
      }
      String getPromotion (Long umE) {
          for (Spécialité sp : spécialité) {
                promo = sp.getPromotion (numE);
                if (promo != null) return promo;
          }
       return null;
}}
```

```
class Etudiant_M1  extends Etudiant {
      String getPromotion () {
          return "Master 1ère année" ;
      }
```

```
class Spécialité {
      String getPromotion (Long numE) {
          for (etu : étudiants) {
              if (etu.getNuméro () == numE)
                    return etu.getPromotion ();
          }
          return null;
```

Principes de conception

☞ Application « Université » : *getMoyenne* avec respect des principes …




```
class Université {
  float getMoyenne (Long umE) {
    for (Spécialité sp : spécialité) {
      try{
        moy = sp.getMoyenne (numE);
        return moy;}
      catch (EtudNonTrouveEx) {}
    }
    throw new EtudNonTrouveEx();
```

```
class Spécialité {
   String getMoyenne (Long numE) {
    for (etu : étudiants) {
      if (etu.getNuméro () == numE)
          return etu.getMoyenne ();
     }
     throw new EtudNonTrouveEx();
```

```
class Etudiant_L1 extends
Etudiant {
   float getMoyenne () {
      return this.noteTC;
   }
}
```

```
class Etudiant_L2 extends Etudiant {
   float getMoyenne () {
      return (etud.getNoteTC() +
            +etud.getNoteOption())
            /2;
 }
```

Principes de conception
OCP

- OCP : Open-Closed Principle
  - Constat : ajouter du nouveau code est moins coûteux que de modifier du code existant !
  - Modifier d'un code existant présente des risques (régression fonctionnelle !)
  - Mécanisme : coder les traitements particuliers dans des classes dédiées
- Exemple

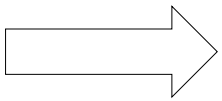
```
class Etudiant {
    String annee;
    …
    float getMoyenne () {
        if (annee.equals ("L1")
                return this.noteTC;
        else if (annee.equals ("L2")
                return (this.noteTC+
                        this.noteOpt)/2;
    }
}
```

Maintenance :
Prise en compte de L3 !
Coût :
Modifier la classe Etudiant (--)

Principes de conception
OCP

☞ OCP : Open-Closed Principle

```
class Etudiant {
    String annee;
    …
    float getMoyenne () {
        if (annee.equals ("L1")
                return this.noteTC;
        else if (annee.equals ("L2")
                return (this.noteTC+
                        this.noteOpt)/2;
    }
}
```



```
interface Etudiant {
        float getMoyenne ();
}
```

```
class Etudiant_L1
implements Etudiant {
        float noteTC;
        float getMoyenne () {
            return this.noteTC;
        }
}
```

```
class Etudiant_L2 imp Etudiant {
        float noteTC;
        float noteOpt;
        float getMoyenne () {
            return (this.noteTC+
                    this.noteOpt)/2;
        }
}
```

Maintenance : prise en compte de L3 !

Coût : coder une nouvelle classe (++)